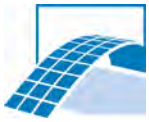




TUM Department of Civil, Geo and Environmental Engineering
Chair of Computational Modeling and Simulation
Prof. Dr.-Ing. André Borrmann

# Automated retrieval of shared IFC model data based on user-specific requirements

**Deian Stoitchkov**

Master's thesis
of the Master of Science program Civil Engineering

| | |
|---|---|
| Author: | Deian Stoitchkov |
| Student ID: | [redacted] |
| Supervision: | Prof. Dr.-Ing. André Borrmann |
| | Sebastian Esser, M.Sc. |
| Date of Issue: | 15. May 2020 |
| Date of Submission: | 15. December 2020 |

## Abstract

At the core of Building Information Modeling (BIM) lie models that are designed to contain large amounts of both semantic and geometric information for a given construction project that can be used throughout the entire life cycle of a built facility. This inevitably requires the use of multiple software applications from different vendors. Therefore, buildingSMART International (bSI) has developed the Industry Foundation Classes (IFC) standard for an open, vendor-neutral exchange of data models. Since the full IFC schema targets a wide range of different use cases and requirements, Model View Definitions (MVD) specify a subset of the IFC schema and are used to certify software applications for the import and export of IFC models. However, the official MVDs provided by bSI are still considered too general for many tasks and must be extended for more specific exchange scenarios. BIM models usually become very large and contain huge amounts of information, of which in many cases only a part is needed. Therefore, this thesis proposes a method for retrieving IFC model data based on user-defined requirements. The user criteria are validated against the official concept templates defined in the mvdXML format by bSI and simplified by using the RuleIDs of certain attributes. The result is a new valid IFC model that contains only the information requested by the user. Furthermore, as multiple IFC models are often created for a single project, it is also advantageous to be able to merge multiple IFC files into a single final model that contains only a part of the information from the original models. A Common Data Environment (CDE) provides an environment to store and manage BIM models for improved collaboration among project participants. Thus, a tool is developed for this work that allows the user to select models directly from a CDE to retrieve specific data. The Python library IfcOpenShell is used to interact with IFC data stored in a STEP Physical File. It is a powerful library providing the necessary tools for reading and modifying existing IFC models as well as creating new models. Autodesk BIM 360 Docs is used in this thesis as a representative of the many CDE platforms currently available. In addition, the Autodesk Forge APIs and services are used to automatically access and retrieve data from BIM 360. Finally, a Graphical User Interface (GUI) is created specifically for this purpose, allowing the user to easily interact with the tool.

## Zusammenfassung

Den Kern von Building Information Modeling (BIM) bilden Modelle, die so konzipiert sind, dass sie große Mengen sowohl semantischer als auch geometrischer Informationen für ein bestimmtes Bauprojekt enthalten, die während des gesamten Lebenszyklus der gebauten Anlage genutzt werden können. Dies erfordert zwangsläufig den Einsatz mehrerer Softwareanwendungen von verschiedenen Anbietern. Daher hat buildingSMART International (bSI) den Industry Foundation Classes (IFC)-Standard für einen offenen, herstellerneutralen Austausch von Datenmodellen entwickelt. Da das vollständige IFC-Schema auf eine Vielzahl unterschiedlicher Anwendungsfälle und Anforderungen abzielt, spezifizieren die Modellansichtsdefinitionen (MVD) eine Teilmenge des IFC-Schemas und werden zur Zertifizierung von Softwareanwendungen für den Import und Export von IFC-Modellen verwendet. Die von bSI bereitgestellten offiziellen MVDs werden jedoch für viele Aufgaben immer noch als zu allgemein angesehen und müssen für spezifischere Austauschszenarien erweitert werden. BIM-Modelle werden in der Regel sehr groß und enthalten riesige Mengen an Informationen, von denen in vielen Fällen nur ein Teil benötigt wird. Daher wird in dieser Arbeit eine Methode zum Abrufen von IFC-Modelldaten vorgeschlagen, die auf benutzerdefinierten Anforderungen basiert. Die Benutzerkriterien werden gegen die offiziellen ConceptTemplates validiert, die vom bSI im mvdXML-Format definiert wurden, und durch die Verwendung der RuleIDs bestimmter Attribute vereinfacht. Das Ergebnis ist ein neues gültiges IFC-Modell, das nur die vom Benutzer angeforderten Informationen enthält. Da für ein einzelnes Projekt oft mehrere IFC-Modelle erstellt werden, ist es außerdem von Vorteil, mehrere IFC-Dateien zu einem einzigen endgültigen Modell zusammenführen zu können, das nur einen Teil der Informationen aus den Originalmodellen enthält. Common Data Environments (CDE) bieten eine Umgebung zum Speichern und Verwalten von BIM-Modellen für eine verbesserte Zusammenarbeit zwischen den Projektteilnehmern. Daher wird für diese Arbeit ein Werkzeug entwickelt, das es dem Benutzer ermöglicht, Modelle direkt aus einem CDE auszuwählen, um bestimmte Daten abzurufen. Die Pythonbibliothek IfcOpenShell wird zur Interaktion mit IFC-Daten verwendet, die in einer STEP Physical Datei gespeichert sind. Es handelt sich dabei um eine leistungsfähige Bibliothek, die die notwendigen Werkzeuge zum Lesen und Modifizieren bestehender IFC-Modelle sowie zum Erstellen neuer Modelle bereitstellt. Autodesk BIM 360 Docs wird in dieser Arbeit als Beispiel für die vielen derzeit verfügbaren CDE-Plattformen verwendet. Darüber hinaus werden die APIs von Autodesk Forge verwendet, um automatisch auf die Daten von BIM 360 zuzugreifen und diese abzurufen. Dies wird über eine speziell für diesen Zweck entwickelte grafische Benutzeroberfläche durchgeführt.

# Contents

# Acronyms

| | |
|---|---|
| **3D** | Three Dimensional |
| **AEC** | Architecture, Engineering, and Construction |
| **API** | Application Programming Interface |
| **BIM** | Building Information Modeling |
| **BREP** | Boundary Representation |
| **bSI** | buildingSMART International |
| **CAD** | Computer-Aided Design |
| **CDE** | Common Data Environment |
| **CPU** | Central Processing Unit |
| **CSG** | Constructive Solid Geometry |
| **GUID** | Globally Unique Identifier |
| **GUI** | Graphical User Interface |
| **HTML** | Hypertext Markup Language |
| **IAI** | International Alliance for Interoperability |
| **IDE** | Integrated Development Environment |
| **IDM** | Information Delivery Manual |
| **IFC** | Industry Foundation Classes |
| **IOS** | IfcOpenShell |
| **KIT** | Karlsruhe Institute of Technology |
| **MVD** | Model View Definition |
| **SPF** | STEP Physical File |
| **URL** | Uniform Resource Locator |
| **XML** | Extensible Markup Language |

# List of Figures

# List of Listings

# Chapter 1

# Introduction

## 1.1 Motivation

In the Architecture, Engineering, and Construction (AEC) sector, Building Information Modeling (BIM) is one of the most promising developments. An accurate virtual model of a building is digitally created with BIM technology. This model can be used for the planning, design, construction and operation of the facility. It helps architects, engineers and designers to visualize what needs to be built in a simulated environment in order to identify potential design , construction or operational problems (Azhar, Salman, 2011). This leads to a more effective and cost-efficient workflow. Therefore, the BIM methodology is currently being adopted by many companies around the world. The maturity of BIM can be divided into different levels, each of which uses specific exchange data and information depth (Figure 1.1).

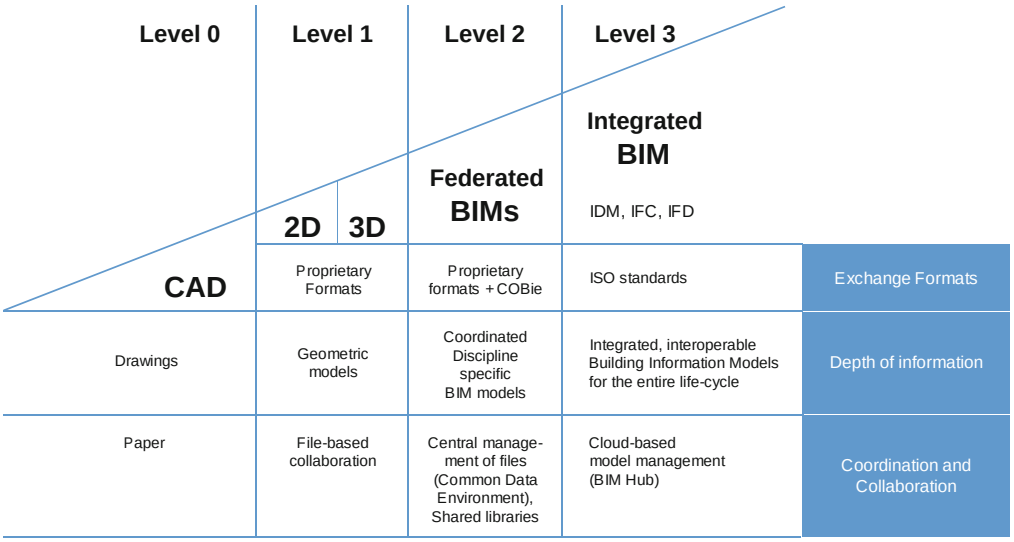



**Figure 1.1:** The BIM Maturity Ramp of the UK BIM Task Group defines four discrete levels of BIM maturity (Bew & Richards, 2011, Borrmann *et al.*, 2018)

BIM's objective is to implement BIG Open BIM, which includes software products from various providers using open data exchange formats, and the continuous utilization of digital building models across multiple disciplines and life cycle phases (Borrmann *et al.*, 2018). For this purpose the use of ISO standards such as the open file format Industry Foundation Classes (IFC) is necessary. BIM models can get very complex and contain information that is not needed for a specific use or workflow. Therefore, the concept of Model View Definition (MVD) was developed, which represents a subset of the entire IFC data model and does not contain all the information that can be represented in a BIM model. An MVD specifies the end user requirements of needed information. There are official MVDs developed by buildingSMART International (bSI) that are used by software applications to become certified to support the import or export of IFC data models. However, these MVDs are usually too generic for specific needs and contain unnecessary information. Therefore, the ability to further filter IFC models with user-defined criteria is required. There are currently several solutions for this purpose, but they are generally overly complicated and require a deep understanding of the IFC schema and often involve the use of multiple software programs.

The ultimate goal is to use cloud services to handle project data so that it is continuously and consistently maintained throughout the life cycle of a building. Currently, however, BIM models are often stored in Common Data Environment (CDE) platforms that contain multiple files for a single project, often filled with information that is too extensive and complex for specific use cases. As a result, there is a need for a user to be able to filter the required data and obtain this data from multiple files simultaneously. This work aims at doing so by allowing a user to filter multiple files stored in a CDE and as a result obtain a single file containing only the requested information from all files. For this purpose, the IFC models are also checked for the presence of the required data, so it is also useful to validate models for the existence of specific information that may be required for further data exchange.

## 1.2 Structure of work

Chapter 2 contains background information on the open exchange format Industry Foundation Classes (IFC). The basic concepts for understanding the structure of an IFC file are explained, such as the IFC schema, the inheritance hierarchy of the object-oriented data model, the relationship between entities as well as the geometric representations and placement of objects. The file types for storing the information are described together with an example of the most commonly used STEP Physical File (SPF). Furthermore, the necessity and purpose of the Model View Definition (MVD) and the use of mvdXML to define allowable values in the IFC file are discussed. The official information provided by buildingSMART International (bSI) for IFC version 4 and later is also described in Chapter 2.

The current situation and similar work is discussed in Chapter 3. The functionalities and limitations of the Autodesk Revit IFC Exporter are described. Furthermore, the current workflow of model validation against an mvdXML is explained and the existing possibilities for filtering IFC models are discussed.

In chapter 4 an approach is proposed for the filtering and merging of multiple IFC files and the way of its implementation. First the choice of programming language and libraries is explained, as well as the software used for this work. For a better understanding of the working process, key terms, such as early and late binding are clarified. Next the procedure for automatically extracting the information from the official bSI website with Python is explained in detail. The structure of the user-defined XML file is described in this chapter, together with its requirements and limitations. The criteria in the XML file can be defined in two different ways: direct access to the attributes of an entity in a hierarchical structure or automatic retrieval of the attributes from an mvdXML concept template using the RuleID of an attribute. The process of checking an IFC file against the custom XML file is described next, along with how the corresponding elements are being filtered and, if available, merge multiple IFC files into a single final result. IFC files can be retrieved automatically from a CDE, the workflow of which is also explained in chapter 4.

The functionality of the tool created for this work is demonstrated in Chapter 5. Example models with the most important entities and properties is presented. The user-defined requirements written in an XML file are discussed, and finally the resulting models are shown.

In the last chapter 6 the limits of this work are discussed. In this chapter a summary of the proposed solution is presented together with an overall conclusion and outlook into the future.

# Chapter 2

# Background information

## 2.1 Industry Foundation Classes

### 2.1.1 General

The processes in the Architecture, Engineering, and Construction (AEC) sector are very complex and take many years involving sometimes even thousands of participants. The life cycle of a building comprises multiple phases such as design, construction and maintaining. This requires the coordination of many different disciplines that often work together and interfere with each other, for which in itself great amount of information flow is needed.

In today's modern world, digital tools are increasingly being used for the design, construction and operation of buildings and infrastructure facilities, as this often proves to be more productive. This is where Building Information Modeling (BIM) comes into play. With BIM it is possible to translate the plan, design, construct and manage of buildings and infrastructure in a intelligent 3D model-based process. What makes such a model very powerful is that it contains not only the geometric data of building elements, but also describes the relationships between them and the definition of their relevant properties such as the materials, fire rating etc. This greatly increases the transfer of information between multiple stakeholders, leading to an improvement in productivity by reducing the laborious and error-prone manual re-entry of information that dominates traditional paper workflows as noted in Borrmann *et al.*, 2018.

In such a complex process, the use of several software programs, often from different providers, is unavoidable. However, the issue is that many of these tools have no or only insufficient support for the sharing of data between different applications. This results in the need for an open, vendor-neutral file format, such as the Industry Foundation Classes (IFC). The purpose of IFC is to allow a standardised exchange of models between different stakeholders

performing various calculations and simulations across the entire life cycle of a built facility (see Figure 2.1).

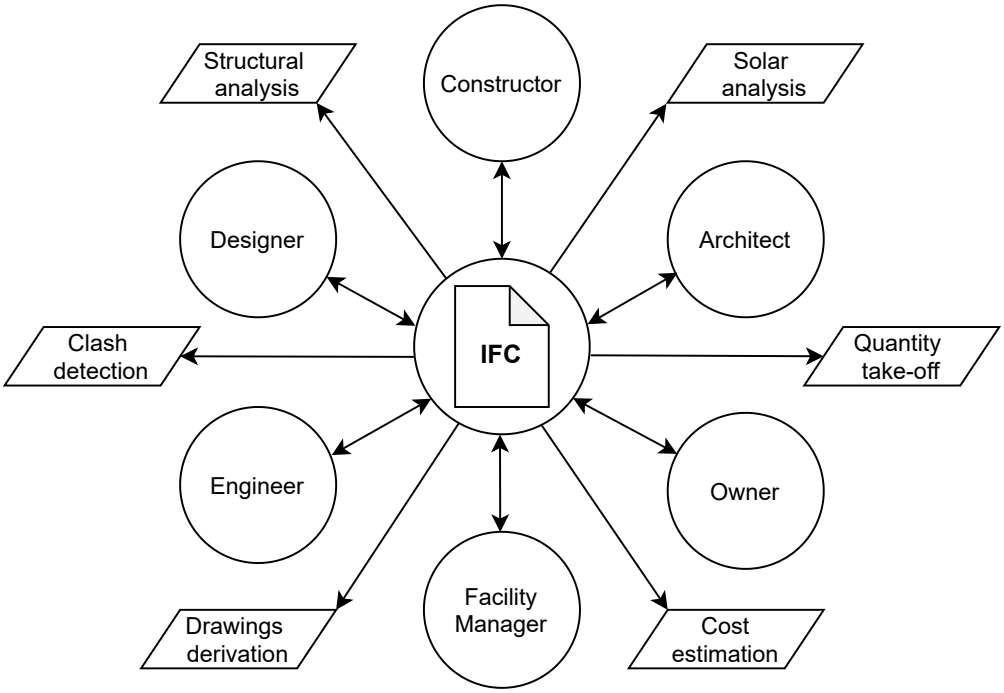



**Figure 2.1:** IFC as a single exchange format for multiple disciplines and participants

The first tools for transferring data between different CAD systems were developed in the 1970s. However these formats were meant solely for the exchange of geometric data, but it was soon realized that a more complex exchange format is required. As a result, the International Alliance for Interoperability (IAI), currently known as buildingSMART International (bSI), was formed to develop and support standards for the entire industry. As a result the first version of Industry Foundation Classes (IFC) was issued in 1997 (Borrmann *et al.*, 2018). Since then, there have been 6 major versions of the IFC, with IFC2x3 and IFC4 currently being used most frequently. IFC5 is currently in the early planning phase, with full support for different infrastructure areas and more parametric functionality planned.

### 2.1.2 Schema

The complex IFC model is divided into four layers for an easier maintenance and future extension. The layers from bottom to top are: resource, core, interoperability and domain layer (see Figure 2.2). The main idea is that elements in a specific layer can only refer to an elements on the same or lower layer, but not the ones on the higher layers.

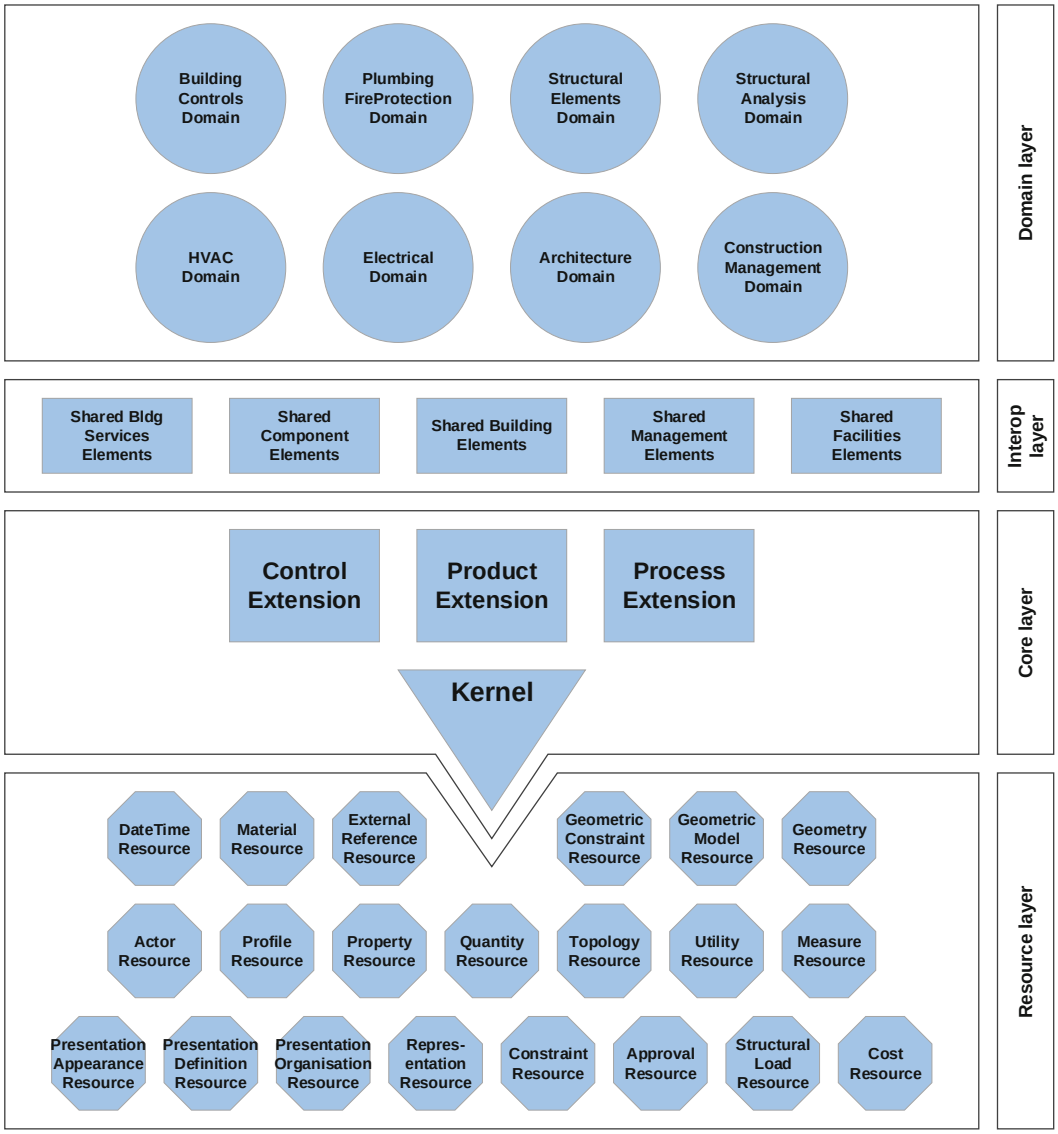



**Figure 2.2:** Layers of the IFC data schema (buildingSMART International, 2020a)

The lowest layer, the resource layer, contains all the individual resource definition schemas that can be used throughout the entire IFC data model. These definitions are not to be used separately of a definition declared at a higher layer, meaning that they can't exist as independent objects in an IFC model and are not derived from the *IfcRoot* entity like all other layers. The most important resource schemes include the geometry resource, topology resource, geometric model resource and the material resource schemes, used and described in more detail further in this work.

The core layer includes the kernel schema, containing the most general entity definitions, the understanding of which is important for this work. All the layers above can reference the

entities in the this layer. The Kernel schema represents the core of the IFC data model and comprises basic entities such as *IfcRoot*, *IfcProject*, *IfcObject*, *IfcProduct* etc. The information provided by *IfcProject* includes the default units, as well as the geometric representation context such as the the project coordinate system and the precision used within the geometric representations. Such information is important for the correct placement and geometric representation of objects, described in Section 2.1.5. The Core Layer also contains the ControlExtension, the ProductExtension and the ProcessExtension. ProductExtension, which describes the physical and spatial objects, is important for this work, which contains subclasses of *IfcProduct* like like *IfcBuilding*, *IfcBuildingStorey* and *IfcSpace* (Borrmann *et al.*, 2018).

The next layer up, the interoperability (or shared elements) layer, contains entities derived from the core layer. Building elements can be found in this layer such as *IfcWall*, *IfcColumn*, *IfcWindow*, etc., defined in five different so-called SharedElements. The Domain Layer is the highest layer and contains highly specialized classes such as electrical systems, heating, ventilation etc. No other layer can reference the classes specified in this layer, as it is at the top of the hierarchy.

### 2.1.3 Inheritance

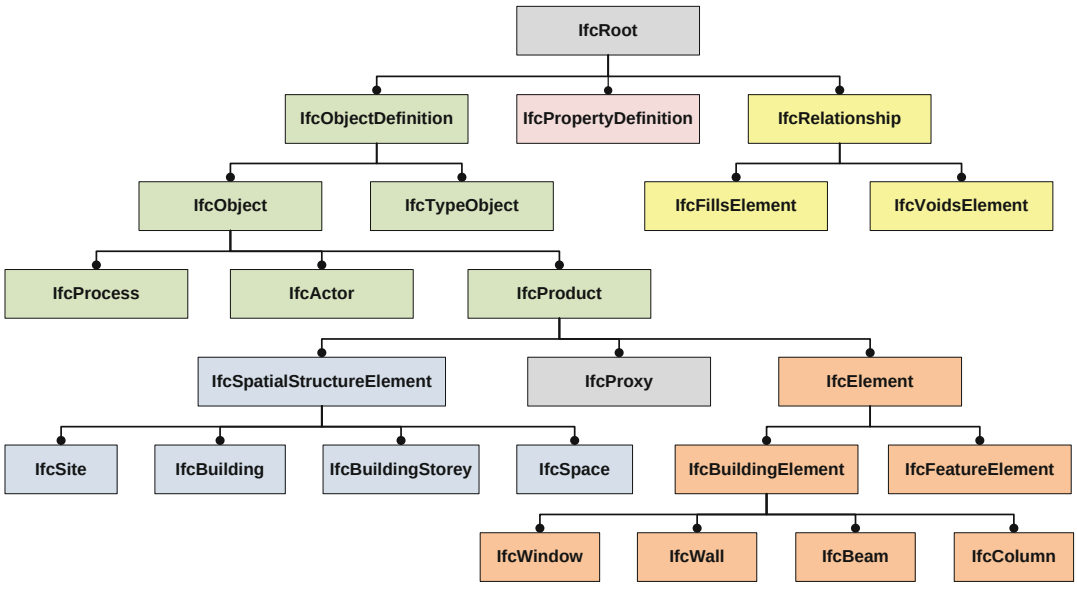



**Figure 2.3:** Part of the inheritance hierarchy of the IFC data model (Borrmann *et al.*, 2018)

One of the most important features of an IFC is its object-oriented data model, including the inheritance hierarchy between different elements. This allows objects at lower hierarchical

levels to receive attributes from their parent objects, making the definition of classes clearer and with less redundancy, since all attributes of a given class can be used from its child classes without having to be redefined.

Some of the most important entities of the inheritance hierarchy are shown in Figure 2.3. It can be seen that all of these entities inherit from the *IfcRoot* entity. As explained earlier, this is not valid for the entities in the resource layer. *IfcRoot* provides important basic information about an entity such as the GlobalId, which uniquely identifies each entity within the entire software world (buildingSMART International, 2020j). As this is the leaf node in the hierarchy all classes displayed in Figure 2.3 have a GlobalId.

The official information provided by buildingSMART International, 2020a contains data about the inheritance of attributes of each entity, including the inverse attributes from the objectified relationships (explained in the next section). This is important because if it is being looked directly at the attributes of a particular entity, important information defined in the classes from which the entity inherits is missing. For example, if we look at the entity *IfcObject*, the inheritance would be:

$$IfcRoot \Rightarrow IfcObjectDefinition \Rightarrow IfcObject$$

From this follows that all attributes defined in *IfcRoot* and *IfcObjectDefinition* can be accessed also within the *IfcObject*. For each entity a well structured documentation is provided by bSI.

| Attribute | Type |
|---|---|
| **IfcRoot** | |
| GlobalId | IfcGloballyUniqueId |
| OwnerHistory | IfcOwnerHistory |
| Name | IfcLabel |
| Description | IfcText |
| **IfcObjectDefinition** | |
| HasAssignments | IfcRelAssigns |
| Nests | IfcRelNests |
| IsNestedBy | IfcRelNests |
| HasContext | IfcRelDeclares |
| IsDecomposedBy | IfcRelAggregates |
| Decomposes | IfcRelAggregates |
| HasAssociations | IfcRelAssociates |
| **IfcObject** | |
| ObjectType | IfcLabel |
| IsDeclaredBy | IfcRelDefinesByObject |
| Declares | IfcRelDefinesByObject |
| IsTypedBy | IfcRelDefinesByType |
| IsDefinedBy | IfcRelDefinesByProperties |

**Table 2.1:** Attribute inheritance of IfcObject (buildingSMART International, 2020b)

An example for *IfcObject* presented in buildingSMART International, 2020b can be seen in Table 2.1 (further information such as cardinality and description of each attribute is also provided, but for simplicity only the attribute name and type are shown here). In this case the *IfcObject* entity possesses all the attributes from *IfcRoot* and *IfcObjectDefinition* together with its own attributes. This holds true for all entities in the inheritance structure (Figure 2.3).

Each entity is assigned a type, which in turn is also an IFC entity. Some of the attributes are simple types like strings (*IfcLabel*, *IfcText*), while others represent inverse attributes describing complex relationships between entities, like the *IfcRelAssociates* or *IfcRelDefinesByProperties*, the latter describing how property set definitions and objects relate to one another. These relationships are an important part of the IFC data model and are explained in the next section.

### 2.1.4 Object relationships

In the IFC the concept of objectified relationships is used. This means that there is one entity that defines the relationship between other entities. Such an entity can be recognized by the "Rel" part in its name, e.g. *Ifc**Rel**AssociatesMaterial*, a sub-type of *IfcRelAssociates* (see Table 2.1), which defines the relationship between *IfcObjectDefinition* and *IfcMaterial*.

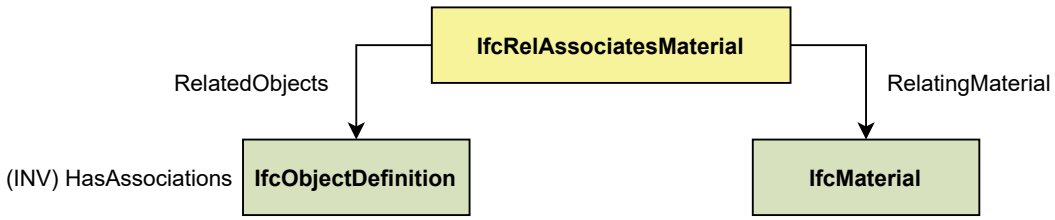



**Figure 2.4:** Relationship between entities

In Figure 2.4 the intermediate entity *IfcRelAssociatesMaterial* has the attribute RelatedObjects, which contains all entities having a certain material, given by the RelatingMaterial attribute. The inverse attribute HasAssociations can be used to navigate the reverse path from the RelatedObjects to the RelatingMateiral, or in other words, it can be used to access the material of the entity *IfcObjectDefinition* (Bim supporters, 2020). In general, this applies to all relationships. Due to the inheritance hierarchy of the IFC data model, all entities that inherit from *IfcObjectDefinition* have the attribute HasAssociations. Figure 2.3 shows that all entities down to *IfcWall*, *IfcColumn* etc. have this inverse relationship.

Another essential objectified relationship is the *IfcRelContainedInSpatialStructure*, which is used to assign elements to a specific spatial object. For example, *IfcWall* can be related to

*IfcBuildingStorey*, which means that the wall is located on a certain floor of the building. Here the intermediate entity has the attribute RelatedElements, similar to RelatedObjects, which contains all entities in this spatial object. It is important to note that an element can only be assigned to one spatial object.

### 2.1.5 Geometric representations

So far, this chapter has explained the semantic description of an IFC. It is not required for a building element to have a defined geometry and a model can be purely semantic (Tobiáš, Pavel, 2015). However, a BIM model would not be complete without the ability to define the geometric representations related to an element. All geometric representations are located in the resource layer of the IFC schema, meaning that they are not derived from IfcRoot and cannot exist as independent objects in an IFC model (see Section 2.1.2). Furthermore, all geometric items used within a representation are derived from the abstract super-class *IfcGeometricRepresentationItem* (buildingSMART International, 2020c). A 3D model can be represented as a surface or solid model, both having their own advantages and disadvantages.

Triangulated meshes are commonly used as a surface model to represent geometry. Almost all software programs supporting IFC can read this basic type of geometric representation, which makes it perfect for a general model used by multiple software vendors. The key drawbacks, however, are that curved surfaces are not represented precisely, but rather only approximated by triangles (tessellation). This indicates that the geometry of the model is not of high quality and a very large number of triangles are sometimes required to represent a 3D body, making it not the most efficient way to construct geometries (Borrmann *et al.*, 2018). Such a geometry is described in the IFC with the *IfcTessellatedFaceSet* class.

Implicit and explicit are the two primary categories for describing solid geometric models. Constructive Solid Geometry (CSG) is an example of an implicit model. It uses a combination of Boolean operations (union, intersection, difference) applied to 3D bodies to construct complex geometries. This results in a tree structure, which contains the construction history of the CSG model, enabling an easy modification of the model later on. The problem with the implicit models is that complex geometric operations need to be calculated in a specific manner from the receiving software, resulting in a more complicated implementation. Therefore not all programs support implicit geometries. However, the smaller file size makes it ideal for big models, frequently seen nowadays.

In contrast to the implicit models, the Boundary Representation (BREP) - an example for an explicit solid model - contains all of the information required to represent the geometric model as explicit boundaries in the IFC file. Such information includes surfaces, edges and vertices with the corresponding coordinates without the need for further calculation, as is the case with the CSG construction tree. Some drawbacks of this model include the larger

file size, however generally still more compact than the tessellated models, and the unknown generation process, which in turn makes it difficult to change this type of geometry. In general it is easily possible to convert an implicit model to an explicit model. The opposite way is more challenging and limited, however, numerous solutions were proposed, such as Buchele *et al.*, 2004 and Shapiro *et al.*, 1993.

### 2.1.6 Object placement

For each object that has a shape representation an *IfcObjectPlacement* has to be defined, which is an abstract super-type for the special types defining the object coordinate system (buildingSMART International, 2020d). There are three ways to represent the object placement: *IfcGridPlacement*, *IfcLinearPlacement* and the *IfcLocalPlacement*. The last of these is the most widely used, which defines the relative placement of an *IfcProduct* in relation to the placement of another such object. A wall with the *IfcLocalPlacement* is not placed using global coordinates, but instead relative to the coordinate system of a *IfcBuildingStorey*, if it lies in a storey. The following conventions apply to the spatial elements that define a spatial structure according to buildingSMART International, 2020e, which are important for the understanding of the relative object placement:

- ***IfcSite*** shall be placed absolutely within the world coordinate system established by the geometric representation context of the *IfcProject*.
- ***IfcBuilding*** shall be placed relative to the local placement of *IfcSite*.
- ***IfcBuildingStorey*** shall be placed relative to the local placement of *IfcBuilding*.

With the *IfcLinearPlacement*, first introduced in IFC4x1, the placement and axis direction of the object coordinate system is defined by a reference to a curve such as *IfcAlignmentCurve*. This is very effective for positioning elements for linear construction works, such as roads, rails, bridges, and others.

### 2.1.7 File types

The IFC data model mainly relies on the data modelling language called EXPRESS. As such, the IFC file is stored in the exchange file format defined in part 21 of the STEP standard ISO 10303 with the .ifc file extension. Another way to store an IFC file is to use XML format, having the file extension .ifcxml. The ifcXML file contains the same IFC information, the only difference being the way the data is described. However, ifcXML files tend to be significantly larger than a regular IFC files having the same data, making the STEP Physical File (SPF) the preferred way for exchanging model data, which is also used in this work. The IFC data

may also be compressed within a ZIP file with the file extension .ifcZIP, resulting in a smaller file size.

## 2.2 Example of a STEP physical file

A STEP Physical File (SPF) file is divided in two sections. The *HEADER* is the first part, which contains general information about the IFC file. This includes the IFC schema version, the Model View Definition (MVD) and the application which was used for exporting the file along with the the date and time of the export (Liebich, T., 2009). An example of the *HEADER* section of an IFC file, exported from Autodesk Revit 2021, is shown in Listing 2.1. Each SPF starts with a line indicating that the file is specified in ISO 10303 part 21. The end of each expression is marked with a semicolon. Line 2 indicates the beginning of the *HEADER* section, line 6 ends it. In between, the actual *HEADER* information is described. The *FILE_DESCRIPTION* contains the formal definition of the underlying view definition (Karl-Heinz Häfele *et al.*, 2008). After that the *FILE_NAME* displays the file name, creation time, author, organization, name of the toolbox and the application used, and the author of the IFC file. Values can also be left empty, which can be placed as empty quotation marks. The IFC version can be found at the end of the *HEADER*.

```
1 ISO-10303-21;
2 HEADER;
3 FILE_DESCRIPTION(('ViewDefinition [ReferenceView_V1.2]'),'2;1');
4 FILE_NAME('SampleFile.ifc','2020-10-21T11:21:49',(''),(''),'The EXPRESS Data Manager Version
      5.02.0100.07','Exporter 21.1.0.0','');
5 FILE_SCHEMA(('IFC4'));
6 ENDSEC;
```

**Listing 2.1:** HEADER section of a STEP physical file

The actual IFC model data is in the *DATA* section of the SPF file (Listing 2.2). Each instance of an entity is displayed on a new line, prefixed by a # character and a unique number, allowing each entity to be referenced by other entities. The object relationships as well as the geometric representations and object placement, which are described in the previous sections, are all described here. It is important to note that Listing 2.2 is only a part of a full SPF and some information is missing for the sake of simplicity. There is an *IfcProject* describing the default units (# 109) along with the *IfcGeometricRepresentationContext* (#114) in the *DATA* section, the use of which is mandatory to represent the 3D model view. Since *IfcProject* is located in the Kernal of the IFC schema (see Section 2.2), it is derived from the *IfcRoot* and has a GlobalId, which uniquely identifies this entity. The *IfcSite* (#161), *IfcBuilding* (#138) and *IfcBuildingStorey* (#151) are represented next, each with a local placement denoted by the *IfcLocalPlacement* (#160, #33 and #149 respectively).

```
1  DATA;
2  ...
3  #6= IFCCARTESIANPOINT((0.,0.,0.));
4  #32= IFCAXIS2PLACEMENT3D(#6,$,$);
5  #33= IFCLOCALPLACEMENT(#160,#32);
6  #42= IFCOWNERHISTORY(#39,#5,$,.NOCHANGE.,$,$,$,1603272486);
7  #43= IFCSIUNIT(*,.LENGTHUNIT.,.MILLI.,.METRE.);
8  #44= IFCSIUNIT(*,.LENGTHUNIT.,$,.METRE.);
9  #109= IFCUNITASSIGNMENT((#43,#45,#46,#50,#52,#55, ...));
10 #111= IFCAXIS2PLACEMENT3D(#6,$,$);
11 #112= IFCDIRECTION((6.12303176911189E-17,1.));
12 #114= IFCGEOMETRICREPRESENTATIONCONTEXT($,'Model',3,0.01,#111,#112);
13 #123= IFCPROJECT('3mP2wl77X9vuLv_35IW4_d',#42,'0001',$,$,'Project Name','Project Status',(#114)
       ,#109);
14
15 #138= IFCBUILDING('3mP2wl77X9vuLv_35IW4_c',#42,'',$,$,#33,$,'',.ELEMENT.,$,$,#134);
16 #148= IFCAXIS2PLACEMENT3D(#6,$,$);
17 #149= IFCLOCALPLACEMENT(#33,#148);
18 #151= IFCBUILDINGSTOREY('3mP2wl77X9vuLv_36jVx5W',#42,'Level 1',$,'Level:8mm Head',#149,$,'
       Level 1',.ELEMENT.,0.);
19 #261= IFCRELCONTAINEDINSPATIALSTRUCTURE('3Zu5Bv0LOHrPC10066FoQQ',#42,$,$,(#187)
       ,#151);
20 #159= IFCAXIS2PLACEMENT3D(#6,$,$);
21 #160= IFCLOCALPLACEMENT($,#159);
22 #161= IFCSITE('3mP2wl77X9vuLv_35IW4_b',#42,'Default',$,$,#160,$,$,.ELEMENT.,(42,21,31,181945)
       ,(-71,-3,-24,-263305),0.,$,$);
23
24 #168= IFCLOCALPLACEMENT(#149,#167);
25 #175= IFCSHAPEREPRESENTATION(#118,'Axis','Curve3D',(#174));
26 #182= IFCPRODUCTDEFINITIONSHAPE($,$,(#175,#243));
27 #187= IFCWALL('161akpRoD3Fv0DQZ79Tc2G',#42,'Basic Wall:Generic - 200mm:346616',$,'Basic Wall:
       Generic - 200mm',#168,#182,'346616',.NOTDEFINED.);
28 #202= IFCMATERIAL('Default Wall',$,'Materials');
29 #213= IFCMATERIALCONSTITUENT('Layer',$,#202,$,'Materials');
30 #218= IFCMATERIALCONSTITUENTSET('Basic Wall:Generic - 200mm',$,(#213));
31 #239= IFCEXTRUDEDAREASOLID(#233,#238,#20,8000.);
32 #243= IFCSHAPEREPRESENTATION(#120,'Body','SweptSolid',(#239));
33
34 #265= IFCRELAGGREGATES('30PQja$0fBXx84$o4cZPDT',#42,$,$,#123,(#161));
35 #269= IFCRELAGGREGATES('3xYfaATBL1VeB5eW8zrLUB',#42,$,$,#161,(#138));
36 #273= IFCRELAGGREGATES('27PCKGLxT4mxtV9cw6mgBW',#42,$,$,#138,(#151));
37 #281= IFCRELASSOCIATESMATERIAL('3_iLtDJQX6L8Z3t$XKEfh0',#42,$,$,(#187),#218);
38 ...
39 ENDSEC;
40 END-ISO-10303-21;
```

**Listing 2.2:** Part of the DATA section of an IFC file

There is a certain relationship between the spatial structure elements, as explained in 2.1.6. The objectified relationship between the *IfcSite* and *IfcProject* is denoted by #265 with the *IfcRelAggregates* relationship, which points to #123 and #161. The same applies to the relationship of *IfcBuilding* to *IfcSite* (#269) and finally *IfcBuildingStorey* to *IfcBuilding* (#273).

The instance of an *IfcWall*, identified with #187, is placed locally in a *IfcBuildingStorey* (#151). Since this is also an objectified relationship it cannot be observed directly neither on line 27 nor on line 18, but an additional entity *IfcRelContainedInSpatialStructure* (#261) is needed to describe the relationship. In this case there is only one entity related to the *IfcBuildingStorey*. However, it is possible and often the case that several entities are linked to a single spatial structural element. The *IfcWall* defined on line 27 also points to an *IfcProductDefinitionShape* (#182), which allows for multiple geometric shape representations of the same product. A specific geometric representation is denoted by *IfcShapeRepresentation*. In this case there are two geometries associated with the *IfcWall*. The first one with a "Curve3D" representation type (#175) and the second one being a "SweptSolid" pointing to an *IfcExtrudedAreaSolid* (#239), which is a representation of a solid model.

As described in Section 2.1.4 materials are specified using the relationship class *IfcRelAssociatesMaterial* linked to a building element. This can be observed on line 37 with the objectified relationship connecting the *IfcWall* (#187) to a material. However, here the relationship is not directly to an *IfcMaterial*, but instead to an *IfcMaterialConstituentSet*, which allows the assignment of multiple individual materials to specific parts of an element.

The structure of the SPF file is very complex, and the complete file must be available in order for the relationships to be interpreted correctly. It is also not easy to make changes, as modifying only one instance of an object may also lead to unwanted changes in the remaining data. Therefore, several tools exist that provide the functionalities for the modification of an SPF taking into account all complex relationships, which are discussed in the next chapters.

## 2.3 Model View Definitions

### 2.3.1 General

The complete IFC schema is very complex and difficult for software vendors to implement correctly, since many hundreds entity definitions and thousands of attributes together with the objectified relationships and geometry representations must be correctly described. It is not only difficult, but also unnecessary as the exchange of models is often required for a specific use or workflow and do not need all the information. Therefore, subsets of data can be defined by parsing the entire IFC schema into smaller "model views" that specify end-user requirements for the information needed. For this purpose buildingSMART International

(bSI) had developed the concept of Model View Definition (MVD), which defines a subset, or a "filtered" view, of the overall IFC schema, required for a specific use. The objects, geometric representations, relationships and attributes that are needed for a specific task are described in an MVD (buildingSMART International, 2020f).

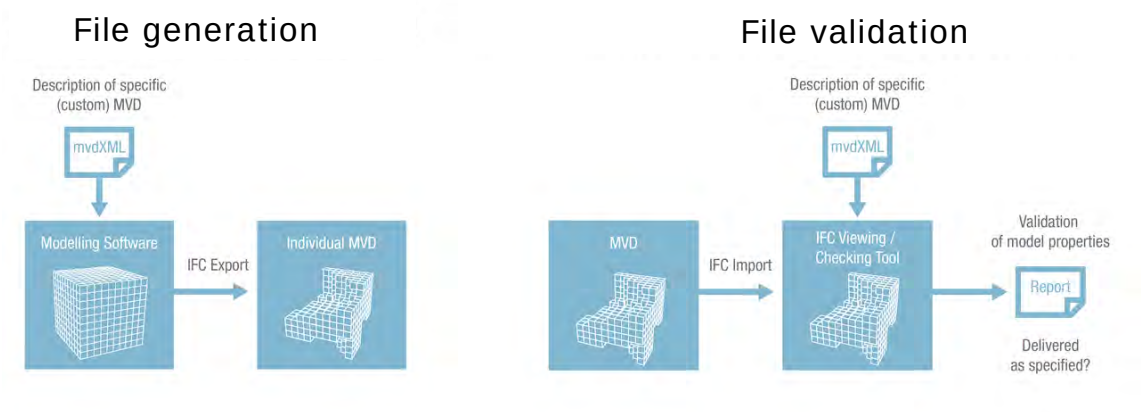



**Figure 2.5:** MVD for generation and validation of IFC files (Baldwin, M., 2019)

MVDs are not only important for the correct interpretation of IFC files from software vendors, but also for validation of existing IFC files against user-defined criteria (see Figure 2.5). To be able to describe and display the information required as a part of the IFC schema, bSI developed the Information Delivery Manual (IDM). The IDM was created to solve collaboration problems in construction projects by capturing business processes and providing user-defined specifications about what data should be delivered, when and by whom. These exchange requirements also specify which data part of the scheme is needed to fulfill such an exchange (Chipman *et al.*, 2016). As such, IDM is essentially an agreement on the processes and responsibilities of the project partners, whereas an MVD clarifies the data implementation details (Weise *et al.*, 2017). Once the knowledge of experts is collected and structured in a human-readable form, it is translated into a machine-readable technological solution based on a reusable collection of IFC concepts. In an IDM requirements come from the needs of the end user and the primary role of Model View Definition (MVD) is to ensure that IFC implementations support those requirements (Hietanen & Final, 2006).

An MVD can be represented in a mvdXML form, which is a machine readable Extensible Markup Language (XML) file describing the exchange requirements. The idea behind it is that software vendors can implement the support for mvdXML and use this mechanism to create IFC import and export filters and to define validation rules (Figure 2.5). MvdXML is based on so called Concept Templates that define how a certain piece of data must be represented in IFC. The structure of such a file is described in detail in the next section 2.3.2.

For reliable and consistent data exchange the MVDs are important part of software certification. Software programs can be certified for specific MVDs, thus allowing it to correctly import and export IFC files without the need for implementing the whole IFC schema, which often includes irrelevant information for the particular use of the software. Thus bSI provides official MVDs, which can be exported and imported from software applications. According to buildingSMART International, 2020g as of writing these are:

| IFC Schema | MVD Name | Summary |
|---|---|---|
| IFC2x3 TC1 | Coordination View | Handover of model information from planning and design applications to CAFM and CMMS applications, as well as the handover of model information from construction and commissioning software to CAFM and CMMS applications |
| IFC2x3 TC1 | Space Boundary Addon View | Identification and export of additional Space Boundaries (polygons which define the extents of a space's contact with directly adjacent surfaces [e.g. walls, floors, ceilings] and openings). Can be used for building energy analysis and quantity take-off. |
| IFC2x3 TC1 | Basic FM Handover View | Handover of model information from planning and design applications to CAFM and CMMS applications, as well as the handover of model information from construction and commissioning software to CAFM and CMMS applications |
| IFC2x3 TC1 | Structural Analysis View | The structural analysis model, created in a structural design application by a structural engineer to one or many structural analysis applications. |
| IFC4 ADD2 TC1 | Reference View | Simplified geometric and relational representation of spatial and physical components to reference model information for design coordination between architectural, structural, and building services (MEP) domains |
| IFC4 ADD2 TC1 | Design Transfer View | Advanced geometric and relational representation of spatial and physical components to enable the transfer of model information from one tool to another. Not a "round-trip" transfer, but a higher fidelity one-way transfer of data and responsibility |

**Table 2.2:** Official Model View Definitions (buildingSMART International, 2020g)

In Table 2.2 only the current official MVDs are represented. There are more MVDs for the IFC4 currently under development such as the Quantity Takeoff View, Energy Analysis View, Product Library View and the Construction Operations Building. It is not required for a software to be certified for both import and export of a specific MVD. This makes it useful for programs that require external model data to preform simulations or calculations, but does not provide export functions.

### 2.3.2 MvdXML

As the name suggests, mvdXML relies on the Extensible Markup Language (XML) that defines a set of document rules in a human-readable and machine-readable format in text-based file. The purpose of this type of file is to store and transport data. What makes it versatile is the ability to create custom elements with attributes in a hierarchical structure. XML does not use predefined tags, so custom tags can be created, making it possible to describe all different kinds of data.

A simple example of an XML structure is presented in Listing 2.3. It describes a building with two floors, each containing additional information. The tags, surrounded by an opening (<) and closing (>) sign (*Building*, *FirstFloor* etc.) describe what the elements are. Each element requires an opening and a closing tag, regardless of how much information is stored in between. Empty elements that have neither text content nor sub-elements can be closed with a self-closing tag, like the one in line 4, using the "/>" symbol. The names of the attributes can also be specified arbitrarily, with each attribute having a value surrounded by quotation marks. On line 1 *type* is the name of an attribute having the value "Office". It is also possible for elements to have multiple attributes and sub-elements.

```
1  <Building type="Office" location="Germany">
2    <FirstFloor>
3      <Height>2.8</Height>
4      <FirstWall type="External"/>
5      <SecondWall type="Internal"/>
6    </FirstFloor>
7    <SecondFloor>
8      <Height>3.2</Height>
9      <FistColumn material="Concrete"/>
10     <SecondColumn material="Steel"/>
11   </SecondFloor>
12 </Building>
```

**Listing 2.3:** Simple XML example

If an element has only text content (e.g. "2.8" on line 3) without attributes and sub-elements, it can be written as an attribute, resulting in an XML file with the same information. Therefore the element *Height* in line 3 could be written as an attribute to the element *FirstFloor* with an attribute name *Height* and a value "2.8". In general, there are no rules about when to use attributes or when to use elements in XML. However, attributes cannot contain multiple values or tree structures and are not easily extendable for future changes (W3Schools, 2020a). It is often necessary to use elements within other elements, which results in a hi-

erarchical structure. The data defined in Listing 2.3 can be displayed as a tree structure for better visualization (Figure 2.6). The element *Building* is located at the top and has 2 child elements, which in turn each have 3 further child elements. With XML, a relatively complex data structure can be easily stored and read by both humans and computers, which makes it very common today. Therefore this file format was chosen to describe the required information from an IFC schema in mvdXML file format.

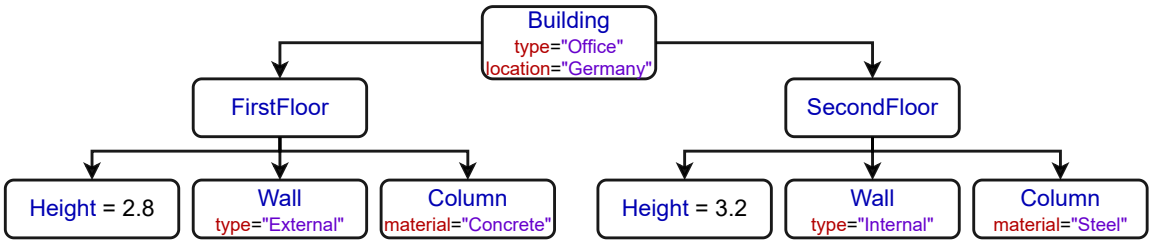



**Figure 2.6:** XML tree structure of the data in Listing 2.3

To describe MVDs and the associated exchange requirements, buildingSMART International (bSI) has developed the standard mvdXML. The mvdXML file structure starts with an *mvdXML* element which has attributes such as *uuid* and *name* and define two main sub-elements: *Templates* and *Views* (Chipman *et al.*, 2016). The *uuid* is a unique identifier that never changes for a particular element. This is an essential part of any mvdXML as it allows an element to be referenced by other elements, so that certain information can be reused several times. Both *Templates* and *Views* elements have no attributes and are used to distinguish between the sub-elements of each element. The tree structure of *mvdXML* and its elements is shown in figure 2.7. The element *Templates* represents a list of several *ConceptTemplates*, while *Views* contains one or more *ModelViews*. Each *ModelView* element describes how *ConceptTemplates* should be used in a particular Model View Definition (MVD). Some notable attributes are the unique identifier (*uuid*) and the IFC schema version to which this MVD applies (*applicableSchema*), along with the name of the model view. It contains the elements (Chipman *et al.*, 2016):

- Definition: a human-readable description of the purpose of generating the MVD documentation.
- BaseView: reference to a base model view definition, if this model view represents an add-on model view that extents a base view
- ExchangeRequirements: a list of exchange requirements defined within this model view.
- Roots: a list of *ConceptRoots*, comprised of concepts, that reference to *ConceptTemplates*.

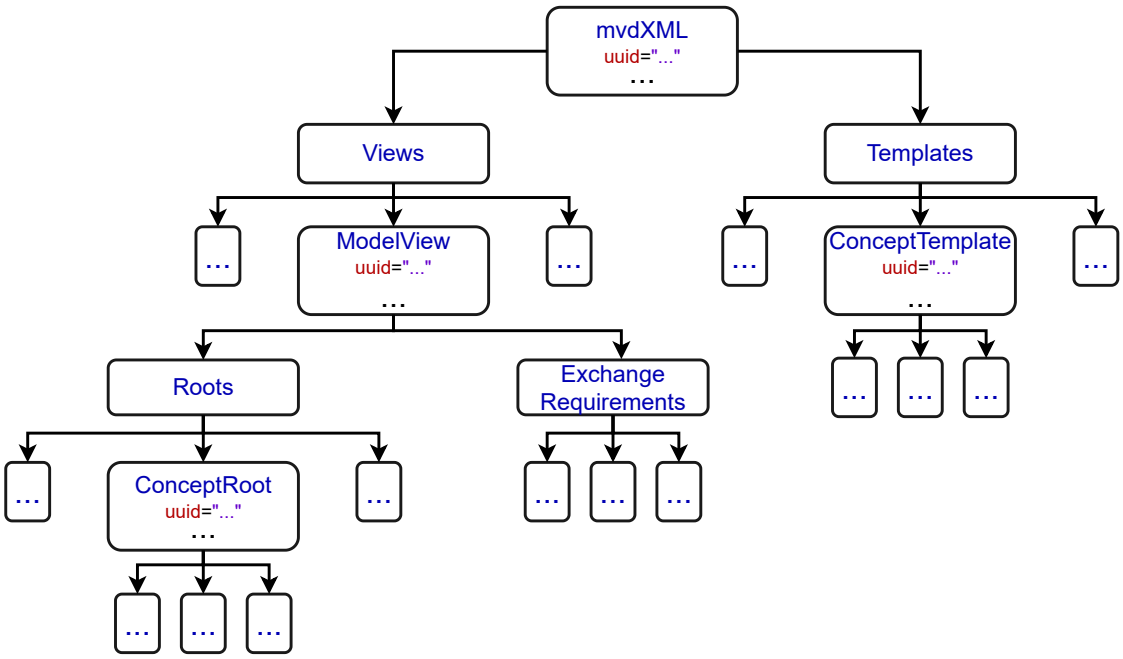



**Figure 2.7:** mvdXML tree structure

*ConceptRoot* refers to the respective IFC entity that is derived from *IfcRoot*. Among others it has an *uuid* and an *applicableRootEntity* as attributes. The latter identifies the class or datatype of the instance that is described or validated, e.g. *IfcWall*. The *Applicability* element within the *ConceptRoot* (see figure 2.8) is a set of *TemplateRules* to specify the applicable instances of the IFC schema. A set of *Concept* elements is defined under *Concepts*, each having:

- Requirements: a list of *Requirement* elements that describes how to manage a concept for each exchange
- Template: references to a *ConceptTemplate* element
- TemplateRules: a tree of *TemplateRule* with a Boolean logic between individual elements (”and”, ”or”).

For this work it is important to understand the path marked in red in Figure 2.8. The hierarchical structure follows: *ConceptRoot* ⇒ *Concepts* ⇒ *Concept* ⇒ *Template*. The last one refers to a *ConceptTemplate*, which consists of attributes and other entity definitions.

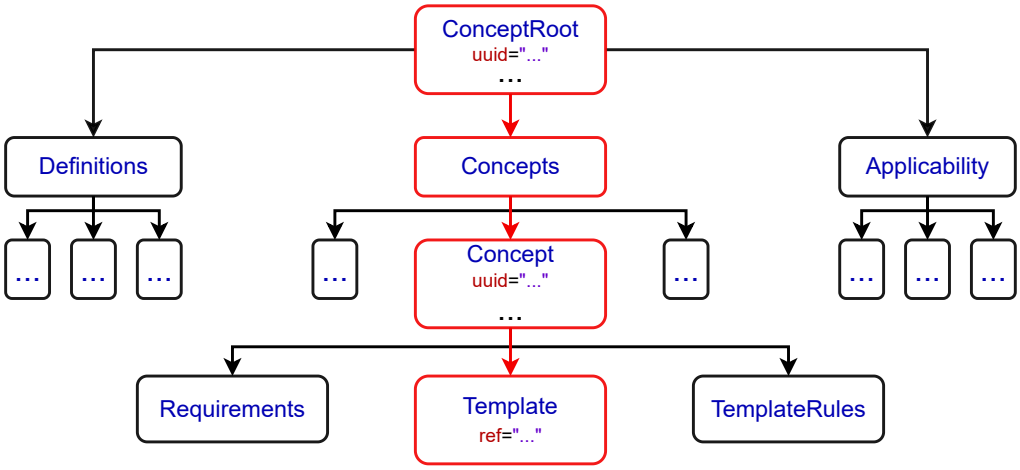



**Figure 2.8:** ConceptRoot tree structure

*ConceptTeplate* describes the reusable concepts as templates. Some of its attributes are the *applicableSchema*, such as IFC2x3 or IFC4, and *applicableEntity*, which includes the entity (e.g. *IfcObject*) to which the concept applies (Chipman *et al.*, 2016). *ConceptTemplate* also has a unique identifier *uuid* as an attribute, which is used to refer to a specific *ConceptTemplate* by a *Template* element within a *Concept* (Figure 2.8). This is done with the help of the *ref* attribute. The child elements of *ConceptTemplate*, which are of importance for this work are:

- SubElements: an optional element that allows multiple *ConceptTemplates* to be grouped under a common criteria.
- Rules: a set of attribute definitions, which relate to the entity defined in the attribute *applicableEntity*

Following the red path in Figure 2.9, it can be observed that *AttributeRule* consists of *EntityRules*, each of which in turn contains *AttributeRules* and so on. This allows to accesses attributes in a hierarchical structure. Each *AttributeRule* element has an attribute *AttributeName*, e.g. *IsDefinedBy* or *GlobalId*. An optional attribute is the *RuleID*, which identifies a specific rule within the tree structure. *EntityRule* contains the attribute *EntityName* that specifies the case-sensitive name of the entity, e.g. *IfcPropertySet*, which can be seen as the type of an attribute in Table 2.1. This is the main principle of an mvdXML and is illustrated in the next section with an example from the official information provided by buildingSMART International (bSI).

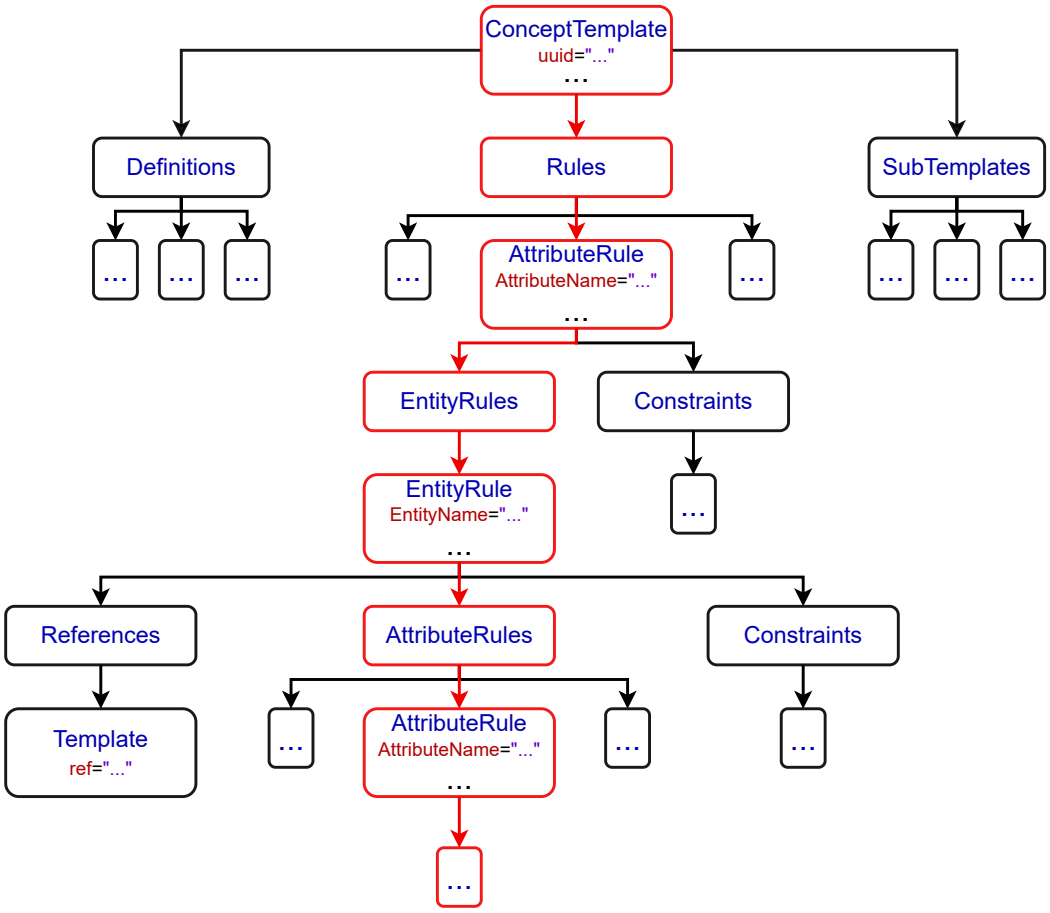



**Figure 2.9:** ConceptTemplate tree structure

### 2.3.3 Official buildingSMART information

buildingSMART International (bSI) provides official information that is accessible to everyone. It contains a detailed explanation of each IFC entity, such as the definition of the entity, the attribute and entity inheritance, and the use of concepts. For IFC version 4 and later, bSI also provides an mvdXML specification, which is a *ConceptRoot* element associated with a particular entity. Furthermore, the mvdXML specifications of *ConceptTemplates* are also given by bSI.

A short example for the mvdXML of an *IfcWall* is presented in Listing 2.4. For simplicity, it is a representation with only a part of the full mvdXML specification provided in buildingSMART International, 2020i, so that certain elements and attributes of elements are missing (this also holds true for Listing 2.5). The mvdXML starts with a *ConceptRoot* element that has a *uuid* attribute along with *name* and *applicableRootEntity* that indicates for which IFC entity can be used. *ConceptRoot* contains an element *Concepts*, which is a list of *Concept* elements. In this case there is a single *Concept* with the attributes *uuid*, *name* and *override*,

which means that the concept is reused without any changes (Chipman *et al.*, 2016). The element *Template* is a reference to a *ConceptTemplate* by *uuid* using the attribute *ref*. In this example, *Template* refers to the *ConceptTemplate* in Listing 2.5 because the value of *ref* corresponds to the *uuid* of the *ConceptTemplate*. As depicted in Listing 2.4, this mvdXML follows the same tree structure highlighted in red in Figure 2.8.

```
1 <ConceptRoot uuid="492da205-54fd-429c-a458-2d7e655f9cbb" name="IfcWall"
      applicableRootEntity="IfcWall">
2   <Concepts>
3     <Concept uuid="c0ad16d6-626a-4af8-9603-387e7e3425b5" name="Quantity Sets"
          override="false">
4       <Template ref="6652398e-6579-4460-8cb4-26295acfacc7" />
5     </Concept>
6   </Concepts>
7 </ConceptRoot>
```

**Listing 2.4:** Simplified part of the mvdXML specification of an *IfcWall*

Listing 2.5 starts with a *ConceptTemplate* element. In addition to the *uuid* and the *name*, the attributes *applicableSchema* and *applicableEntity* exist, which define the IFC schema and IFC entity to which this concept applies. In this case, the *ConceptTemplate* applies to all *IfcObject* entities. Therefore, this concept is also valid for *IfcWall*, since *IfcWall* inherits from *IfcObject* (see Figure 2.3). Next there is the *Rules* entity, which comprises attributes defined at the *applicableEntity*. As illustrated in Table 2.1, the *IfcObject* has an attribute *IsDefinedBy*, which is of type *IfcRelDefinesByProperties*. This relationship is described in Listing 2.5 with *AttributeRule* having the *AttributeName* equal to "IsDefinedBy", which in turn is of the entity type defined by the *EntityName* attribute of the *EntityRule* element. The *IfcRelDefinesByProperties* entity has an attribute *RelatingPropertyDefinition* of type *IfcElementQuantity*, which in turn has an attribute *Name*. As can be observed, this results in a tree structure, in which an attribute of an entity is again an entity, whose attributes can be accessed and so on. This results in the hierarchical structure, which is represented by the red path in Figure 2.9. The *AttributeRules* element can contain several *AttributeRule* elements, which is also valid for the *EntityRules*. As explained previously, *AttributeRule* can also contain a *RuleID* attribute to facilitate the referencing of rules defined in Concepts.

The use of concept templates and the tree structure of an mvdXML makes it a very powerful method to define MVD specific IFC subset schemes describing the required entities and attributes. It is possible to write an mvdXML based MVD with any text editor. But since it tends to become very complex, it is expected that special software applications are used to read and write mvdXML files (Chipman *et al.*, 2016). The use of such software programs is often not simple, since a deep knowledge of the IFC schema is required and the necessary

documentation to create a fully valid mvdXML is often not provided. More information on this is provided in the next chapter.

```
1  <ConceptTemplate uuid="6652398e-6579-4460-8cb4-26295acfacc7" name="Quantity
       Sets" applicableSchema="IFC4" applicableEntity="IfcObject">
2    <Rules>
3      <AttributeRule AttributeName="IsDefinedBy">
4        <EntityRules>
5          <EntityRule EntityName="IfcRelDefinesByProperties">
6            <AttributeRules>
7              <AttributeRule AttributeName="RelatingPropertyDefinition">
8                <EntityRules>
9                  <EntityRule EntityName="IfcElementQuantity">
10                   <AttributeRules>
11                     <AttributeRule RuleID="QsetName" AttributeName="Name">
12                       <EntityRules>
13                         <EntityRule EntityName="IfcLabel" />
14 ...
```

**Listing 2.5:** Simplified part of the mvdXML specification of Quantity Sets

An additional information provided by buildingSMART International (bSI) is an instance diagram for each concept template that visually illustrates the mvdXML for this particular concept. The information from the mvdXML in Listing 2.5 is shown graphically in Figure 2.10. Here it is clearly visible how the objectified relationship *IfcRelDefinesByProperties* connects the entities *IfcObject* and *IfcElementQuantity*. It is important to note that the attributes marked in blue have a *RuleID* defined in the mvdXML in addition to the *AttributeName*. This is the case for the attribute *Name* of the entity *IfcElementQuantity* which is defined in the mvdXML on line 11 in Listing 2.5.

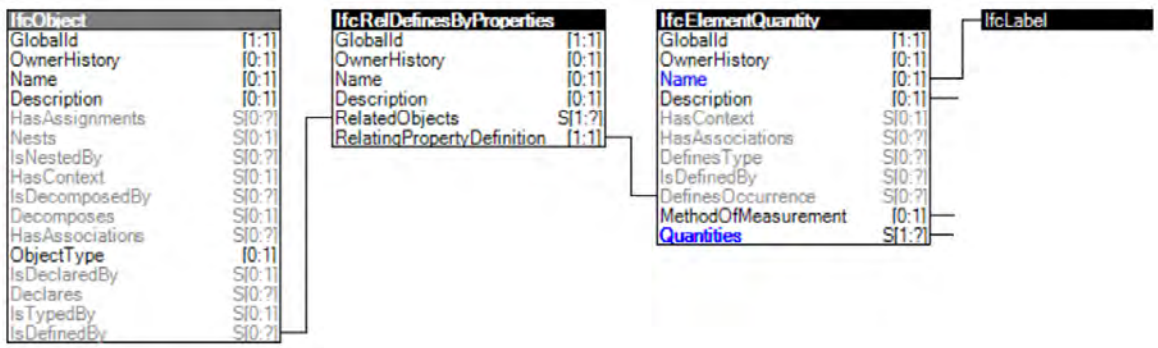



**Figure 2.10:** Part of an instance diagram for Quantity Sets representing the mvdXML in Listing 2.5

# Chapter 3

# Current situation and related work

## 3.1 Generation and export of BIM models

The building model is the key element of BIM technology. Buildings and infrastructure elements can be modeled as virtual representations, which requires a powerful program capable of modeling in a 3D environment, including not only the geometry of the elements but also having a rich semantic information. A number of different companies offer such programs, whose functionalities are similar, but having certain differences that makes each one superior for certain modeling tasks. However, the use of a particular modeling software depends strongly on the personal preference of the user.

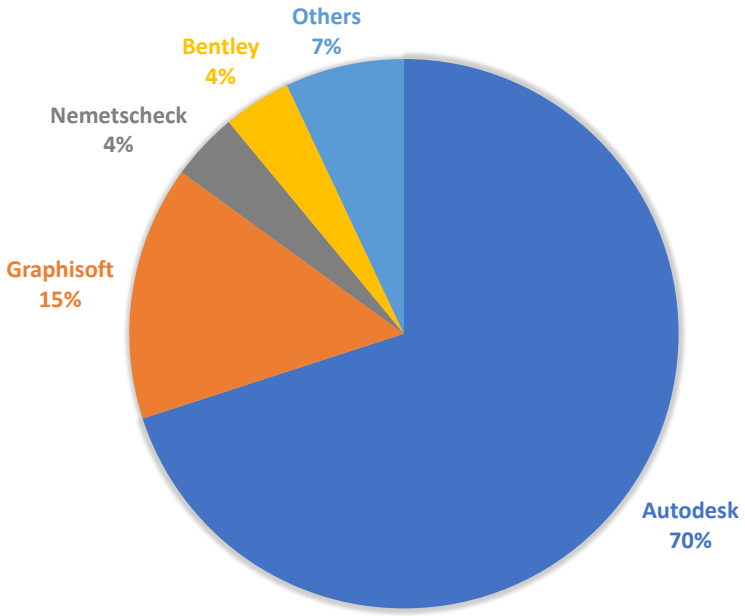



**Figure 3.1:** Most used BIM software vendors for producing drawings and models (NBS National BIM Report, 2019)

According to a survey made in NBS National BIM Report, 2019 the most widely used software applications by their vendors are: Autodesk with 70% of the participants using it, Graphisoft with 15% and Nemetscheck with 4%, as illustrated in Figure 3.1. This survey was conducted in the United Kingdom, but the results also apply to the rest of the world. The modelling software application provided by Nemetscheck is called Allplan, Graphisoft has developed the ArchiCAD and Autodesk offers the Revit software tool.

As mentioned above, Revit is one of the most used applications for building model creation. Models created with Revit can be saved in a native file with the .rvt extension, which is used to store and exchange models between participants. However, this file type is supported only from Autodesk software applications. In large projects, there is often a need to use software tools from different vendors for certain tasks. For this purpose, BIM modeling applications offer the possibility to export and import an IFC file (explained in detail in Chapter 2). The ability to support IFC data models is becoming increasingly important with 77% of participants used IFC for projects in which they were involved in 2019, which is an increase from 2018 according to NBS National BIM Report, 2019. Therefore, Revit supports the export and import of multiple MVDs for the corresponding IFC versions in addition to its native .rvt format, such as the IFC2x3 Coordination View and IFC4 Design Transfer View (see Table 2.2 for more information on the official MVDs).

Software applications can be certified to import and export IFC data models for a specific Model View Definition (MVD). Therefore, an IFC exported from a program is always a filtered view of the IFC and does not necessarily contain instances of every IFC entity available in the full schema. In general, however, only the official MVDs can be exported and only limited customization is possible. The general methodology for exporting an IFC file from Revit is that for each Build-in category in Revit an appropriate IFC entity type is mapped, resulting in a mapping table. The mapping table for IFC imports is structured in a similar way, a part of which can be seen in Figure 3.2.

| IFC Class Name | IFC Type | Revit Category | Revit Sub-Category |
|---|---|---|---|
| IfcAirTerminal | | Air Terminals | |
| IfcAirTerminalType | | Air Terminals | |
| IfcAnnotation | | Generic Annotations | |
| IfcBeam | | Structural Framing | |
| IfcBeamType | | Structural Framing | |
| IfcBoiler | | Mechanical Equipment | |
| IfcBoilerType | | Mechanical Equipment | |
| IfcBuildingElementPart | | Parts | |
| IfcBuildingElementPartType | | Parts | |
| IfcBuildingElementProxy | | Generic Models | |
| IfcBuildingElementProxyType | | Generic Models | |

**Figure 3.2:** Section of Revit mapping table for IFC imports

For example, the Revit category *Structural Framing* is assigned to the IFC entity *IfcBeam*. This table is saved as a text file and can be edited directly in Revit or with any text editor (Autodesk, Inc., 2018). It is also possible to assign generic family instances that are not yet mapped to specific IFC entity types.

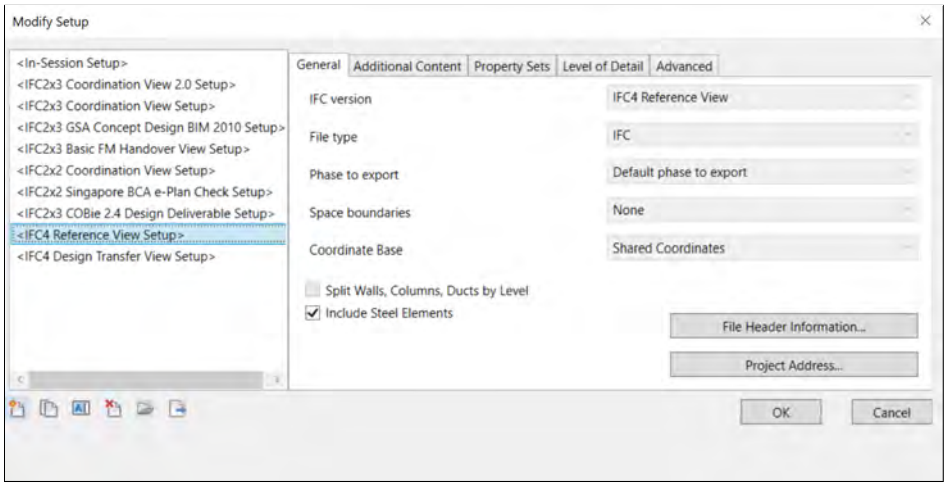



**Figure 3.3:** IFC export options in Revit 2021

The IFC export in Revit can be customized to a certain extent. For this purpose, there is an option to modify the setup for export (see figure 3.3), which allows you to create a new custom setup. This is still based on one of the official MVDs provided by buildingSMART International (bSI), such as the coordination view for IFC2x3 or the reference view for IFC4. The options offered in this menu are important for the richness of the IFC file at the end. There are 5 tabs: General, Additional Content, Property Sets, Level of Detail and Advanced. Some notable options include the ability to export only items that are visible in the view, and whether to export the IFC common property sets or base quantities.

## 3.2 Validate and filter IFC models

All areas of the building industry are addressed by IFC. While this is beneficial for reasons of interoperability, it also contributes to some issues, since this method is too general. Since there are many engineering domains that partly affect each other, there are inevitably optional types of data that in one domain represent essential information, while in another domain they are not needed (Baumgärtel *et al.*, 2016). Therefore, the methodology of Information Delivery Manual (IDM) and Model View Definition (MVD) was developed to define the exchange requirements for models to specify the IFC entities and attributes together with their complex relationships and geometry representations (see Chapter 2.3).

The MVDs reduce the scope of the complete IFC model to subsets that must be supported by export and import of software applications. As shown in the previous section, the user can modify the IFC export, e.g. map native categories correctly to IFC entities or add specific property sets, which can significantly change the information being exchanged. Accordingly, the quality of the exported IFC models by end users is validated by software tools during all life cycle phases of a building to ensure the models carry the required information, in addition to software implementation quality of IFC import and export capabilities being subject to the certification processes defined by buildingSMART International (bSI) (Zhang *et al.*, 2016). Therefore, the exchange requirements can be expressed creating an MVD, which is used to validate the quality of the information provided in an IFC model. An MVD can be specified in a machine readable file named mvdXML, which is described in detail in Chapter 2.3. The exchange requirements are created through declaring classes and attributes. For this purpose, the official software tool for MVDs from bSI ifcDoc can be used (buildingSMART International, 2020k). BIMQ is another platform that enables the definition of MVDs. However, it is important to further improve the instruments for mvdXML generation and validation since most of the available tools do not support mvdXML validation without errors, according to Popgavrilova, G., 2020 thus making the checking results not entirely accurate. There are other commercially available tools such as the Solibri Model Checker, which can perform clash detection and code compliance checking based on predefined rule sets, and Simplebim providing a real time data validation.

Since IFC files tend to become very large and complex, and it can sometimes take hours before they can be simply opened by certain software applications, there is a need to filter the data for future exchange. In general an mvdXML can be used not only for validation but also for filtering data, since it can describe the required information. However, most tools only offer the possibility to validate IFC data models against specific requirements, with no or only limited filtering possibilities. Several approaches for semantic queries for the IFC object model have been introduced, guided by the need to provide tools for convenient information extraction. One of the first query methods for IFC data model was proposed by Adachi, 2003. In this work the Partial Model Query Language (PMQL) was introduced to provide a general method for select, update, and delete partial model data that contains certain part of model data. PMQL allows recursive expressions to be used, thereby reducing the complexity of queries (Tauscher *et al.*, 2016). Another approach, proposed in Weise *et al.*, 2003, is the Generalized Model Subset Definition (GMSD). This method offers the possibility of dynamic selection of object instances in model server queries, defined in the EXPRESS language to be consistent with the IFC data model. The Building Information Model Query Language (BIMQL) can be used for the selection, addition and update of partial aspects in building information models, introduced by Mazairac & Beetz, 2016. This domain specific query language for building models is inspired by PMQL, supporting cascading in order to enable recursive expressions (Tauscher *et al.*, 2016). To this category of querying languages also belongs the Query Language for Building Information Models (QL4BIM), first introduced in

Daum *et al.*, 2014. QL4BIM is unique, since it allows querying spatial information providing metric, directional and topological operators for defining filter expressions with qualitative spatial semantics as stated in Daum & Borrmann, 2014. This allows to query specific building elements, which answer to questions such as: "Which columns touch a certain slab?" (Daum & Borrmann, 2014).

A different approach was proposed in Tauscher *et al.*, 2016, which aims at general information retrieval using graph theory on the IFC object model, where the classes are nodes and the relations between them are displayed as edges. The use of a shortest path algorithm between nodes allows to determine the associations between objects automatically. Retrieving specific information from BIM models using visual languages is introduced in Preidel *et al.*, 2017. The visual query language (vQL4BIM) and the visual code checking language (VCCL) are used, which provide operators to allow handling of relations. Further filtering possibilities for BIM models were also discussed in Windisch *et al.*, 2012 and Wülfing *et al.*, 2014.

As shown, a lot of work is dedicated to the development of a system for retrieving specific information from BIM models. This demonstrates the challenge of this task, since there is no single or best solution and it is still an ongoing topic today. As stated in Preidel *et al.*, 2017:

> *"It is very likely that the importance of Data Retrieval features will continue to grow, as the increased use of BIM in practice results in more and more engineers and architects applying this method. Furthermore, greater use will make the projects, the digital models and project structures more complex. For this reason, the demand for tools architects and engineers can easily use to extract appropriate information from the models will be higher".*

# Chapter 4

# Solution and implementation

## 4.1 IFC libraries: early binding vs. late binding

As explained in Chapter 2.2, the STEP Physical File (SPF) is very complex, therefore a powerful tool is required to create or modify this type of file. There are several tools for interacting with the IFC model, each offering similar functionality but often using different programming languages to utilize it. The general methodology of using these tools is shown in Figure 4.1. A SPF can be read, created and modified using an IFC library to create an application for a specific task. Xbim is a .NET open source software development BIM toolkit that supports Industry Foundation Classes (IFC) (Xbim Toolkit, 2020a). It allows to read, create and view IFC models with support for geometric, topological operations and visualisation.

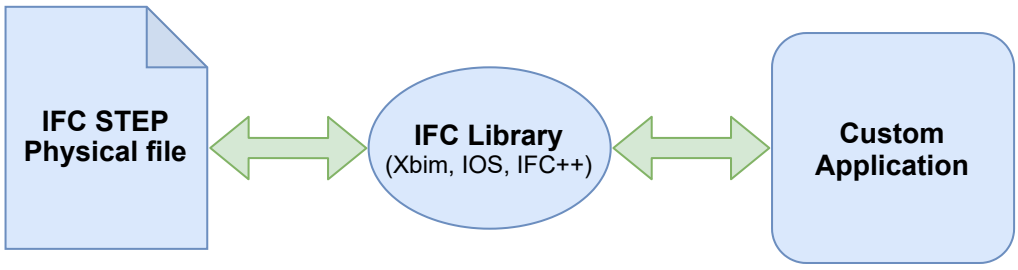



**Figure 4.1:** Methodology of using IFC libraries

The Xbim functionalities are accessed using the programming language C#. A simple example of how to use Xbim is shown in Listing 4.1. An IFC file with the name "SampleHouse.ifc" is read and the information is accessed. Queries can easily be handled with Xbim, like the one on line 4 which retrieves all entities of a certain type, in this case *IfcDoor*. All instances of this entity are then stored in a variable and can be retrieved later in the program.

```
1 const string fileName = "SampleHouse.ifc";
2 using (var model = IfcStore.Open(fileName))
3 {
4     var allDoors = model.Instances.OfType<IIfcDoor>();
5 }
```

**Listing 4.1:** An example for an early binding approach for querying *IfcDoor* entities using Xbim with C# (Xbim Toolkit, 2020b)

Another tool for working with IFC files is IfcOpenShell (IOS). It is also an open source software library that helps users and software developers to work with the IFC file format (IfcOpenShell, 2020a). It is a powerful tool that provides everything needed to access and modify IFC elements. The main difference to Xbim is that IOS provides a module for the Python language, which is used to present a simple example in Listing 4.2. This example has essentially the same functionality as the one in Listing 4.1, namely opening an IFC file, finding all entities of type *IfcDoor* and storing them in a variable for later use. The examples show how the two libraries differ in the way they are used for the same task.

```
1 fileName = "SampleHouse.ifc"
2 model = ifcopenshell.open(fileName)
3 allDoors = model.by_type("IfcDoor")
```

**Listing 4.2:** An example for a late binding approach for querying *IfcDoor* entities using IfcOpenShell (IOS) with Python

There is an important difference in how the queries are performed in Xbim and IOS. Xbim uses the so-called **early binding** approach, which requires each entity in the IFC schema to be represented as a corresponding class in the programming language, in this case C#. In Listing 4.1 the *OfType* method is used to search for elements by their types. In this case a type *IIfcDoor* has already been defined, which corresponds to the IFC entity *IfcDoor*. This requires the generation of a code that maps the classes from IFC to C# classes, which can be used later. However, when new IFC entities are added or existing ones are changed, the data must be mapped again accordingly. Therefore, a separate early binding is required for each IFC scheme version. (Borrmann *et al.*, 2018). Since the information is predefined and available during programming, a major advantage of the early binding approach is that when using a suitable Integrated Development Environment (IDE) for programming with C#, such as Microsoft Visual Studio, the auto-completion functionality makes it easier to access the required information without having to know all available predefined classes perfectly.

On the other hand, IfcOpenShell is based on the **late binding** approach. What this mean is that this method is used to access entities based on the IFC schema definitions during program runtime, unlike the predefined classes used in Xbim. This can be seen in Listing 4.2,

where the method *by_type* from IOS is used to find all entities of type *IfcDoor*. This method is accepting a string value, which uniquely identifies the IFC entity and must correspond to the official entity type, defined by buildingSMART International (bSI). Therefore, when using a late binding approach, no code generation is required to define classes, making adaptation to changes made to the underlying schema more flexible (Borrmann *et al.*, 2018). However, the IFC entities and attributes are accessed via strings, which can have any value and cannot be checked against a predefined class name. This could lead to syntax errors, since strings are not checked by the compiler and are only visible after the program has been executed.

As shown above, Xbim and IOS both provide a similar structure for entity queries and general operation with IFC files. However, they use different methods, each of which has its own advantages and disadvantages. Therefore it depends on the task to be performed and the personal preference of programming language and environment. However, as of writing IfcOpenSchell looks to be the better supported tool with multiple commits each day to the official GitHub repository (IfcOpenShell, 2020b). This is important, since IFC is continuously evolving and some BIM tools are still couple of years behind the release of the most recent IFC versions. Moreover, since IOS is based on the late binding approach, it is not necessary to write code for each predefined entity, so that the end user of the application can flexibly define the requirements. This makes it possible to program a tool that can be written once and used in a variety of situations and also works for the upcoming changes in the IFC schema.

IFC++ or IfcPlusPlus is also an open source library, which can read and write IFC files in STEP Physical File (SPF) format (IFC++, 2020). As the name suggests, this is a C++ library that is very efficient and thus makes working with large IFC files very fast. However, since speed is not of central importance for the purposes of this work, as no real-time visualization or modification of models is performed, this library was not considered further. Instead IfcOpenShell was chosen for the reasons explained above and also due to the personal preference for the programming language Python.

## 4.2 Software applications and programming languages

As explained in the previous section, IfcOpenShell (IOS) is an open source software library for operation with the STEP Physical File (SPF). It provides everything needed to read, modify and create IFC data models, making it the ideal choice for this work. IOS is based on the late binding approach, which makes it flexible because it relies on strings to access entities during program runtime, rather than predefined classes as in the case of the early binding approach. This has the disadvantage, that strings are not checked by the compiler, which means that all entity type names must be known by the user. However, there is a solution to this problem

by using the official information provided by buildingSMART International (bSI), which is explained in section 4.3.

IfcOpenShell is written in C++ and can be compiled (translated from a human-readable programming code to a machine-readable executable program) using Microsoft Visual Studio (IfcOpenShell, 2020b). In addition, the IOS library is also available as a Python module, which can either be compiled from the source code or downloaded as ready-to-use module. It is available as an Anaconda package and can be installed in a Python environment. In addition, the Python module is officially provided by IfcOpenShell ready to be downloaded, which includes the latest updates found in the GitHub repository (IfcOpenShell, 2020c).

Python is a high-level programming language that makes it easier to write programs with fewer lines of code and offers better code readability compared to similar programming languages available. This makes Python a good choice for almost any purpose and is therefore the programming language most sought after by developers who are not currently using it according to the Stack Overflow 2020 Developer Survey. Python code can be written in any text editor and executed with the interpreter that comes with the installation. However, an Integrated Development Environment (IDE) can be used to speed up programming by providing code completion, error checking and a debugger. In addition, different programming entities are displayed in different colors, typically known as syntax highlighting, which improves the clarity of the written code. One of the most popular Python IDEs is PyCharm, developed by JetBrains, a software vendor specialized in creating intelligent development tools for many programming languages (PyCharm, 2020).

The creation of BIM models requires a powerful tool, that is capable of representing not only 3D geometrical data, but also the associated semantic information. Autodesk is the most popular software provider of BIM tools, with Revit being the first choice for many projects (see Chapter 3). Since Revit is certified for the export and import of many of the official MVDs, it is used in this work to create sample BIM models to test the developed tool. The models are created in the native Revit environment and are exported in a STEP Physical File (SPF) for further use. STEP Physical File file can also be loaded into Revit, but since the main purpose of this tool is not to work with IFC files but with the native Revit .rvt format, there is a lack of available information when importing IFC files.

In Figure 4.2 is displayed a screenshot from the properties panel of an imported SPF in Revit, representing the IFC parameters from an *IfcWall*. The IFC model containing the wall was modelled in Revit and exported as a SPF. As shown, there are only limited properties available to the user without providing important information, such as the spatial containment and relationships. Revit is a modeling tool and as such it offers the possibility to modify the model, but it lacks the functionality to view all information in a IFC data model in a structured way. Therefore, there are software applications designed for the specific task of correctly opening and displaying IFC files.

| IFC Parameters | |
|---|---|
| IfcGUID | 2LMu7dZ993KBO92hparaUL |
| Reference | Interior - 135mm Partition (2-hr) |
| IsExternal | ☐ |
| NameOverride | Basic Wall:Interior - 135mm Partition (2-hr):347985 |
| ObjectTypeOverride | Basic Wall:Interior - 135mm Partition (2-hr) |
| ExtendToStructure | ☐ |
| LoadBearing | ☐ |
| Tag | 347985 |

**Figure 4.2:** Ifc properties for an *IfcWall* displayed in Autodesk Revit

The FZK Viewer was developed by the Karlsruhe Institute of Technology (KIT) and has established itself as an independent open source viewer (Autodesk, Inc., 2018). It supports the file formats STEP Physical File and ifcXML from version IFC2X onwards with properties and relationships between objects that are displayed textually, which improves the readability of information in an IFC (KIT, 2020). Support for the latest IFC4X3 schema is also available, a feature that very few tools offer. The FZK viewer can display all of the information provided in a SPF with not only the direct attributes of an entity, but also the type of geometry representation and the objectified relationships, such as the spatial containment or materials that are linked to a specific entity. Therefore, in this work the FZK-Viewer is used to manually check the correctness of both the geometry and the semantic information of the generated IFC files.

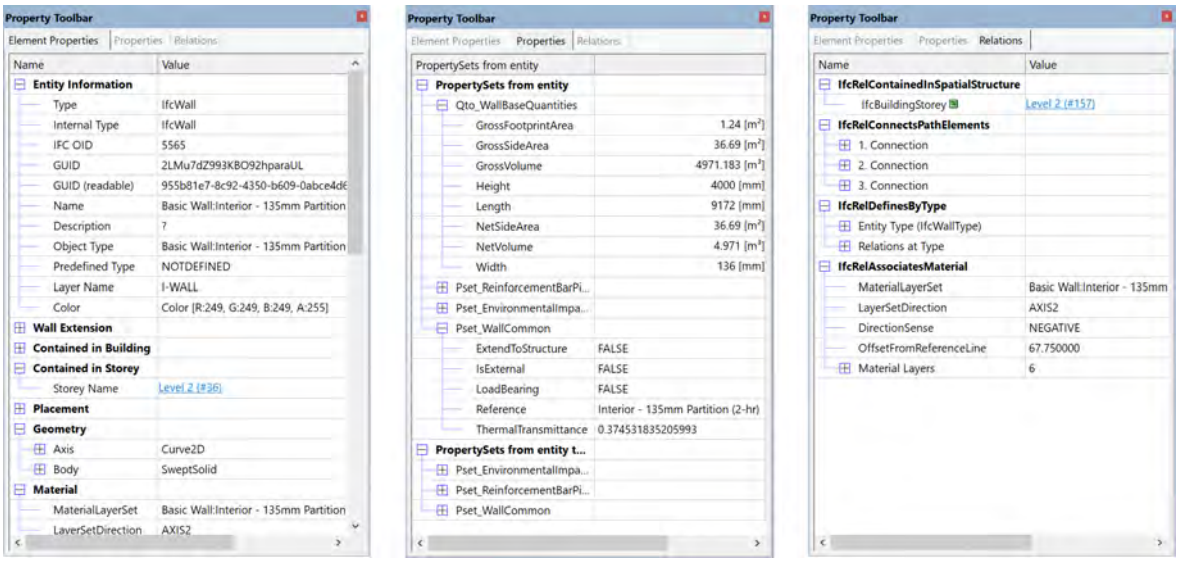



**Figure 4.3:** Ifc properties for an *IfcWall* displayed in FZK Viewer

In addition to the 3D visual representation of the model geometry, the FZK-Viewer offers a property toolbar that displays the information of a specific instance of an IFC entity. Figure 4.3 shows the toolbar, which displays the information of the same *IfcWall* instance as the one in Figure 4.2. This can be seen from the identical Globally Unique Identifier (GUID) in both figures, which corresponds to the GlobalID of an *IfcRoot* and all of the classes that are

derived from it (see Chapter 2). As shown, the FZK-Viewer offers far more information than Revit for the same IFC entity instance. It is well structured and is displayed in the following 3 tabs:

- Element Properties: includes a general entity information, as well as the spatial containment, the geometry representation, the related material and more (Figure 4.3 Left).
- Properties: contains the property and quantity sets with the corresponding individual properties and quantities (Figure 4.3 Centre).
- Relations: shows the objectified relationships associated with this entity instance together with the relating element (Figure 4.3 Right).

Solibri is another tool for viewing BIM models in the SPF format. In contrast to the FZK Viewer, this is a commercial tool developed by Nemetschek (Solibri, 2020). Solibri offers many functionalities to not only display models correctly, but also to check them against predefined rules and to perform collision detection. It is provided in different versions, with Solibri Anywhere being free to use as of writing, which provides everything needed for opening and displaying IFC models with both the geometry representation and the semantic information. However, Solibri does not always correctly display elements from newer IFC versions, such as the *IfcAlignment* added in IFC4x1.

## 4.3 Retrieving the official buildingSMART information

### 4.3.1 General

The data displayed by most websites can only be viewed with a web browser and cannot be stored locally. Therefore, web scraping exist, which is a technique for the extraction of large amounts of data from web sites, whereby the data is extracted and stored in a local file on a computer or in a database in tabular form (WebHarvy, 2020).

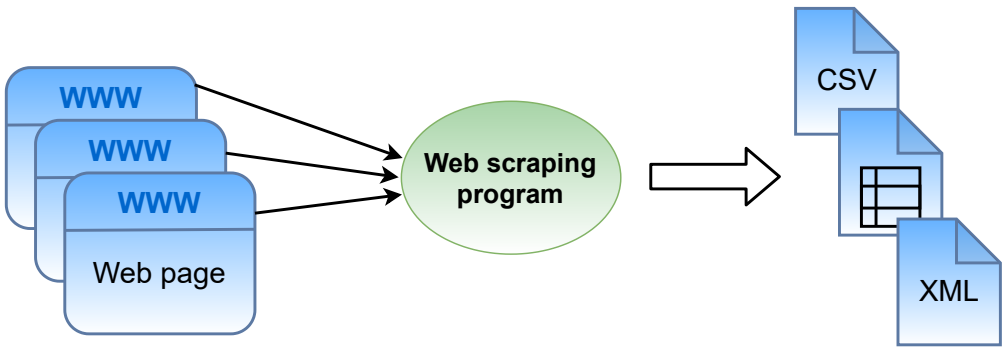



**Figure 4.4:** Web scraping methodology

IfcOpenShell (IOS) is a very powerful library for working with STEP Physical File (SPF), which is constantly being developed and provides support for the latest IFC versions. It is based on the late binding approach as described in section 4.1. This allows for IFC entities to be both instantiated and accessed at runtime (Borrmann *et al.*, 2018). This is done by using string values for each entity name (see Listing 4.2), which creates the flexibility for the end user of the program to access different entities. Since the purpose of this work is to make a general filtering tool that can be used for many of the IFC entities, the use of a late binding approach is chosen, which allows to write a general program that can be operated with user-defined criteria.

However, since the IFC is very complex, it is impractical for the user to define the attributes and relationships of an entity solely on his own. This would be difficult and error-prone process. Therefore, buildingSMART International (bSI) provides official information for each IFC entity class, which is available for free to everyone (see Section 2.3.3). Among others there is data about the inheritance of entities and attributes, as well as an mvdXML specification with a *ConceptRoot* element that describes the concepts for a particular entity (see Section 2.3.2). For each *ConceptTemplate* there is also an mvdXML specification that describes the path for accessing certain attributes with *AttributeRule* elements containing *EntityRules*, which in turn have *AttributeRules* and so on (Figure 2.9). The result is a tree structure that can be used to access nested entities.

This information can be accessed to validate the user-defined requirements against it. This ensures the correctness of the data and allows the user to easily access certain nested properties using the RuleID found in some of the *ConceptTemplate* elements. Therefore, the bSI data provided on the official website is accessed and stored in a structured way that is later used in the developed tool.

### 4.3.2 HTML structure

Most of the websites that are currently online provide information to a user using the Hypertext Markup Language (HTML). HTML describes the structure of a web page with elements that instruct the browser how to present the content (W3Schools, 2020b). These elements are used in a similar way to those in an Extensible Markup Language (XML) file. However, the main difference is that the HTML element tags are predefined, unlike the XML tags, which can be defined by the user.

A simple example of an HTML is presented in Listing 4.3. The file starts with a line telling the browser what type of document it can expect, in this case HTML 5. It is followed by the *html* element, which is a container for all other HTML elements. It has the same general structure as a XML file with opening and closing element tags, where each element can have sub-elements, attributes and text content.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <div class="FirstSection">
      <h1>Heading text</h1>
      <p>Paragraph text.</p>
    </div>
  </body>
</html>
```

**Listing 4.3:** Simple HTML example

HTML provides predefined element tags, each of which has a specific purpose and cannot be changed, such as *head*, *body*, *div* etc. For example, the *div* element has the function of defining a section in an HTML document and is recognized as such by web browsers. The *body* element contains all of the information that is provided to the user, including text, images and links to other web pages, while the *head* element is a container for HTML metadata about the HTML document that is not displayed on the page (W3Schools, 2020b). The *class* attribute of the *div* element specifies a class name of the element that can be accessed by CSS and JavaScript to perform certain tasks. The attribute name cannot be changed and is only used for this purpose, unlike the arbitrarily defined attributes of an XML element. Elements can contain text between the opening and closing tags, as is the case with the *h1* and *p* elements. The element tag represents the type of the element and the way it is displayed by the web browser, while the text is the content that is presented.

The official information provided by buildingSMART International is available on web pages that are linked with one another using hyperlinks. Therefore, for the purposes of this work, the available data is accessed and stored, which is described in the next section. This is done by retrieving the content from various HTML elements and transforming it into structured data that can be used later.

### 4.3.3 Getting and storing the information

The first part of scraping a web page is to access and download the page. To get a page, the Python library *requests* can be used. *HTTP* defines a set of request methods to specify the desired action to be performed on a particular resource (MDN web docs, 2020). These are: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE and PATCH. For

the purpose of this work, only the GET request method from the *requests* library is needed, which requests a representation of the specified resource. If successful, the GET request returns a status code 200, in this case together with the HTML content of the website.

Information for each IFC version from IFC2x on is provided by buildingSMART International under the section *RELEASE*. There is also a *DEV* section which contains information about future versions and currently has a placeholder also for the expected IFC5. This work focuses on the released versions IFC4 and IFC4x1 (buildingSMART International, 2020h). Furthermore, the IFC entities are divided into the schemas contained in the 4 main layers of the IFC schema. The *Shared Building Elements* schema is located in the interoperability layer and contains important building elements including *IfcWall*, *IfcBeam*, *IfcWindow*, *IfcColumn* and *IfcPlate* (see Chapter 2.1.2). On the other hand, the *Product Extension* schema is part of the core layer and also contains essential entities, such as the *IfcElement*, from which all building elements inherit (see Figure 2.3), as well as the *IfcAlignment* that was first introduced in IFC4x1. The present work focuses on these two schemes and the entities contained in them, since they represent some of the most important IFC classes.

To access the data of a particular web page, the correct Uniform Resource Locator (URL) must be passed to the GET request method. Listing 4.4 shows how to retrieve the information provided on the official bSI website. After defining the base URL for all IFC versions under the category *RELEASE*, the IFC version is defined, which in this case is the final version of the IFC4x1. Then the schema is defined, in which the entities to be extracted are contained (line 4). Finally, the name of the entity is specified. All this information is added to a single string that is passed as an argument to the *get* method of the *requests* library. If the request was successful, the function returns the content of the web page together with other information about the performed request and stores the data in the variable *page*, which can be further used to obtain certain information. This process is improved by automatically retrieving all entities from the schemas and extracting the data for each one without having to manually run the program for each individual entity.

```
1 import requests
2 base_url = "https://standards.buildingsmart.org/IFC/RELEASE/"
3 ifc_version = "IFC4_1/FINAL"
4 schema = "ifcsharedbldgelements"
5 entity = "ifcwall"
6 page = requests.get(f"{base_url}/{ifc_version}/HTML/schema/{schema}/lexical
    /{entity}.htm")
```

**Listing 4.4:** Getting the data for *IfcWall* from the official web page

The *get* method of the *requests* library returns information about the request together with the content of the extracted data. However, the content of the website is in an HTML format

from which information is difficult to retrieve. Therefore an additional Python library is used to access the HTML data and extract the required information. The Beautiful Soup library for Python is one of the most popular approaches to this task. It can search for certain HTML tags and retrieve the data contained in them.

As described in the previous section, the visible data of a web page is contained in the *body* tag of an HTML. The *ConceptRoot* definition for some of the most important IFC entities is provided by buildingSMART International. It is contained in the *body* of the HTML under the mvdXML Specification section. The mvdXML is represented within a *code* HTML tag, which is used to define text as computer code. In Listing 4.5, after importing the latest version 4 of the Beautiful Soup Python library, the previously requested page content is specified as a parameter to retrieve a Beautiful Soup object that represents the document as a nested data structure. HTML elements can now be easily accessed, making data extraction process more convenient. The Beautiful Soup object *bs_data* has a method *find_all* that retrieves all of the HTML elements with a specific tag. On line 3 all of the *code* elements are obtained. Since the *code* element containing the mvdXML specification on the bSI web page belongs to the *xsd* class, only the elements of this class are retrieved. After all *code* elements from the *xsd* class are present, the next step is to find the one that has a parent element with a text content "mvdXML specification", since this is the one that is wanted.

```
1 from bs4 import BeautifulSoup
2 bs_data = BeautifulSoup(page.content)
3 html_codes = bs_data.find_all("code", class_="xsd")
```

**Listing 4.5:** Parsing a page with BeautifulSoup

After the *ConceptRoot* for a specific entity has been extracted and saved to a local file, the next step is to retrieve all *ConceptTemplates* applicable to that entity (see Section 2.3.2). The web page of each entity contains information about the concepts, including a hyperlink to the page describing each individual concept template. Such a link is expressed with the HTML element "<a>", which has an attribute *href* with a value corresponding to the URL of the web page. Therefore, with the help of Beautiful Soup, these HTML elements can be accessed and filtered by using the text content that corresponds to the names of the concepts defined in the *ConceptRoot*. On the web page of each *ConceptTemplate* there is an mvdXML specification together with a diagram that visually represents the description of the mvdXML. Therefore, after retrieving the respective URL, the *requests* library is reused to get the HTML content and the process of extracting the *ConceptTemplate* mvdXML is repeated again.

After the script to extract the data for each entity is executed, the information is stored locally in structured form in XML files so that it is easily accessible for each entity later in the program. This includes the *ConceptRoot* definition along with all concept templates that apply to this entity. This process involves a large amount of information that is extracted

repeatedly in the same way for each entity. Since the data for each entity is accessed and stored independently, the process can run in parallel. All modern CPUs have multiple cores that allow several tasks to be processed simultaneously. The *multiprocessing* is a Python package for process-based parallelism. It is used in this work to speed up the process of extracting the buildingSMART International information. A new process is created for each entity, which can run in parallel with other processes, so that the information of several entities can be retrieved and stored simultaneously. The effectiveness depends strongly on the CPU used and the internet connection, but in general the parallel execution of the program is much faster. The program has been tested with the AMD Ryzen 2600x CPU with 6 cores and 12 threads, which is a typical modern processor at the time of writing. It takes 252 seconds (4.2 minutes) to run the program for all entities in the *Shared Building Elements* and the *Product Extension* schemas for both IFC4 Addendum 2 Technical Corrigendum 1 and IFC4x1 Final using a traditional approach, while the multiprocessing method requires only 31 seconds (0.5 minutes). This is more than eight times quicker execution time and significantly reduces the time needed to start the program for the first time, as this process is only performed once to extract the data, after which it is saved for further use.

## 4.4 User-defined requirements

### 4.4.1 General

The goal of this work is to enable a user to define specific custom requirements for retrieving a certain information from a STEP Physical File. A subset of the IFC schema can be defined with a Model View Definition (MVD) as described in Chapter 2.3. MVDs in turn are specified in a machine readable format by creating an mvdXML file. Since the structure of the mvdXML is very complex, there exist tools for creating such files. IfcDoc is one of the tools that can be used to define exchange requirements and formulate MVDs into the mvdXML format (buildingSMART GitHub, 2020).

However, there are often issues associated with creating and exporting the mvdXMLs, as described in Popgavrilova, G., 2020. In addition, the necessary documentation for using the tool is lacking, making it difficult to get started with. Therefore, this thesis proposes a simplified way of accessing entities and attributes, which is still based on the Extensible Markup Language (XML) file, but can be written and used directly by the developed tool. This does not require the use of multiple software applications and reduces the complexity of the entire process. The data downloaded from buildingSMART International is used to validate the user-defined criteria in XML format and also provides easy access to the information described in the *ConceptTemplates* for each entity.

### 4.4.2 Search by Attributes

In an XML file elements can be created with custom tags, that represent information in a meaningful way. Therefore, certain elements are defined for specific purposes to describe the data required from a STEP Physical File. To access a specific attribute, the user can define the path to this attribute in a nested structure. XML is very useful for this, because elements can be nested inside each other. In addition, the XML elements have attributes that provide additional information. More information about the Extensible Markup Language can be found in Chapter 2.3.

The XML file that describes the custom requirements starts with a *Requirements* element that contains the information for each entity that is filtered. The entities are described as sub-elements of the *Requirements* element. It is allowed to have multiple entities in one *Requirements* element, so that multiple IFC entities are filtered from the original SPF. Listing 4.6 provides an example of a user-defined criteria for retrieving the *IfcWall* entity instances. The XML element *Entity* contains all requirements that apply to a specific entity, which is specified in the *Type* attribute. The example shown in Listing 4.6 describes the criteria for filtering all *IfcWall* instances that have a *PredefinedType* of "NOTDEFINED" and a "Brick, Common" *MaterialLayer*, which has a thickness of 90.

```
<Requirements>
  <Entity Type="IfcWall">
    <Attribute Name="PredefinedType" Value="NOTDEFINED" />

    <Attribute Name="HasAssociations">
      <Attribute Name="RelatingMaterial">
        <Attribute Name="MaterialLayers">
          <Attribute Name="Name" Value="Brick, Common"/>
          <Attribute Name="LayerThickness" Value="90.0"/>
        </Attribute>
      </Attribute>
    </Attribute>
  </Entity>
</Requirements>
```

**Listing 4.6:** User-defined requirements for filtering *IfcWall* entities

An attribute of an entity can be accessed with the *Attribute* tag. The name of the IFC entity attribute is defined in the XML *Name* attribute of the element. This allows the user to define any attribute, since the value of a XML attribute can be an arbitrary string. However, since each entity has specific attributes, the user-defined values are validated against the

official information that was downloaded, the process of which is explained in the previous section. A certain Value of an attribute can be specified with the *Value* attribute (lines 3, 8, 9). It is important to note that an attribute can only be assigned a value if it is a string, number (double and integer) or boolean, such as *IfcLabel*, *IfcLengthMeasure* and *IfcBoolean*. However, all values are specified as strings by the user, including boolean and numeric values, so the strings are converted to the corresponding type at runtime. Direct attributes of entities can be accessed in a single line, such as the criteria for the *PredefinedType* specified as "NOTDEFINED" on line 3.

As described in Chapter 2.1.4 the objectified relationships play a crucial role in each STEP Physical File. A material can be assigned to any *IfcWall* with the help of the *IfcRelAssociatesMaterial* entity. This is an objectified relationship, since this entity defines the relation between other entities and can be recognized by the "Rel" part in its name. The inverse attribute *HasAssociations* can be used to access the objectified relationship from the *IfcWall*. The *IfcRelAssociatesMaterial* in turn has an attribute *RelatingMaterial*, which can be used to link a material. In this way, a material can be connected to a wall through the objectified relationship. This principle applies not only to materials, but also to much of the information that can be associated with the entities in the IFC schema, such as property sets, quantity sets, spatial containment, and more.


**Figure 4.5:** Part of an instance diagram for the Material Layer Set *ConceptTemplate* (buildingSMART International, 2020l)

Listing 4.6 provides an example for the definition of a criteria for a specific material layer of an *IfcWall*. XML *Attribute* elements can be nested within each other, so that attributes of other entities can be accessed, which are connected by objectified relationships. The path to access an *IfcMaterialLayer* from an *IfcObjectDefinition*, from which *IfcWall* is derived, is visually represented in Figure 4.5. Such an instance diagram is officially provided for each *ConceptTemplate*. The path to access the material layer is described in an XML format in Listing 4.6. It starts with an *Attribute* element that has a *Name* attribute corresponding to the inverse attribute of the *IfcWall*. This attribute in turn is an *IfcRelAssociatesMaterial* that has an attribute *RelatingMaterial*. The resulting connection is described with the *Attribute* elements, nested in one another. The *Value* XML attribute is used to specify the user criteria in case a certain value of an attribute is required. In this case, certain values are given to the *Name* and the *LayerThickness* of the material layer. An *IfcMaterialLayerSet* consists of several *IfcMaterialLayer* instances, therefore the user requirements for each one must be checked at program runtime until the correct one is found, if such a material layer exists.

### 4.4.3 Search by RuleID

The previous section introduced the methodology for defining user requirements. Information can be retrieved with nested elements, so that complex relationships associated with a particular entity can be accessed. However, this method tends to become overly complex with a number of nested elements that must be described correctly in a XML format.

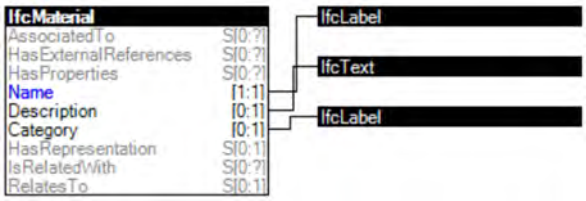



**Figure 4.6:** Instance diagram for the Material *ConceptTemplate* (buildingSMART International, 2020m)

This can be seen in Listing 4.7, where an attribute element has been added in addition to the requirements in Listing 4.6, so that the name of a specific material of a material layer can be defined. Single materials are represented as an *IfcMaterial* instances (Figure 4.6). The user requirements become very complex with many nested elements for each of which the corresponding closing tag must be available. This could lead to errors made by the user, so a different approach is proposed, in which the RuleIDs of certain *AttributeRules* are used (see section 2.3.3 for more information).

```
<Requirements>
  <Entity Type="IfcWall">
    <Attribute Name="HasAssociations">
      <Attribute Name="RelatingMaterial">
        <Attribute Name="MaterialLayers">
          <Attribute Name="Material">
            <Attribute Name="Name" Value="Brick, Common"/>
          </Attribute>
        </Attribute>
      </Attribute>
    </Attribute>
  </Entity>
</Requirements>
```

**Listing 4.7:** User-defined requirements for filtering *IfcWall* entities with specific criteria for the material

The *ConceptTemplates* for all entities, for which the developed tool can be used, were downloaded and stored in a structured way for easy access (see Chapter 4.3). The information

provided by them can therefore be used to validate and simplify user requirements. As described in section 2.3.3, the attributes marked in blue in the instance diagram of a *ConceptTemplate* have a *RuleID* defined in the mvdXML in addition to the *AttributeName*. Since the mvdXML specification is stored for each concept these RuleIDs can be accessed so that the path for accessing certain attributes can be taken from the mvdXML file. This allows the user to specify only the RuleID of a specific attribute without the need to create complex nested requirements.

```
<Requirements>
  <Entity Type="IfcWall">
    <Concept Name="Material Layer Set">
      <RuleID MaterialName="Brick, Common"/>
    </Concept>
  </Entity>
</Requirements>
```

**Listing 4.8:** User-defined requirements for filtering *IfcWall* entities with specific criteria for the material using *RuleID*

In Listing 4.8 a user-defined criteria is presented to get all *IfcWall* from an IFC that have a material with the name "Brick, Common", which is accessed with the RuleID *MaterialName*. This results in the same requirement as the one described in Listing 4.7. This time instead, the RuleID of an *AttributeRule* is used to access it, which is defined in the mvdXML of the *ConceptTemplate*. This leads to a significant reduction of the complexity in defining requirements and thus contributes to the reduction of the errors made by the user. If there are multiple requirements for a single entity, which is often the case, the reduction in complexity is exaggerated by the use of RuleIDs. The RuleID element is placed inside a *Concept* element that defines the concept template containing this particular RuleID. This information is provided by bSI for each entity and can be easily accessed by anyone. In Figure 4.6 the *Name* attribute of the *IfcMaterial* entity is marked in blue, which means that there is a RuleID associated with. This also applies to the *LayerThickness*, *IsVentilated*, *Name* and *Category* of the *IfcMaterialLayer* in Figure 4.5. All of these can be easily accessed with the RuleID of an *AttributeRule*.

Using the RuleIDs from the corresponding *ConceptTemplate* is very useful for defining the requirements. However, since the RuleID is only provided for a limited number of attributes, there is also the need to access attributes manually. For the definition of specific criteria, the complete path with nested elements must therefore still be defined. Furthermore, the name of an entity attribute is often not identical to its RuleID. For example, the RuleID of the attribute *LayerThickness* is the same as the attribute name, but the RuleID of the *IsVentilated* is *AirGap* and can only be accessed with this name (Figure 4.5). However, once the RuleIDs are known, the user requirements can be created very efficiently.

### 4.4.4 Predefined tags and additional functionalities

The XML elements for the purpose of defining user requirements have predefined tag names that serve a specific purpose. Each XML file used in the developed program starts with a *Requirements* element. This element contains all XML elements corresponding to the IFC entities that are filtered by the user. An *Entity* element in turn has a *Type* attribute specifying the entity. In Listing 4.9 on line 2, there is an *Entity* element with a type *IfcWall* that contains all requirements for the instances of this IFC entity. The attribute *Type* can be arbitrary specified for different entities such as *IfcWindow*, *IfcBeam*, *IfcSlab* etc.

```
1  <Requirements SurfaceModel="True">
2    <Entity Type="IfcWall" AllAttr="True">
3      <Attribute Name="PredefinedType" Value="NOTDEFINED[or]USERDEFINED"/>
4
5      <Concept Name="Property Sets for Objects">
6        <RuleID PropertyName="ThermalTransmittance" Value="[>]0.1[and][<]0.5"/>
7        <RuleID PropertyName="IsExternal"/>
8        <RuleID PropertyName="LoadBearing" Value="True" />
9      </Concept>
10   </Entity>
11 </Requirements>
```

**Listing 4.9:** General user-defined requirements for filtering *IfcWall* entities

It is also possible to define requirements for an entity that is a supertype of other entities. For example, the user can request all *IfcBuildingElement* entities from which the elements *IfcWall*, *IfcSlab*, *IfcWindow* etc. inherit. Listing 4.10 shows an XML example that is used to retrieve all building elements that are located on the first floor of the building. However, it is important to note that this would only work if the *IfcBuildingElement* is contained in the spatial structure element *IfcBuildingStorey*.

```
1  <Requirements SurfaceModel="False">
2    <Entity Type="IfcBuildingElement" AllAttr="True">
3      <Concept Name= "Spatial Containment">
4        <RuleID SpatialElementName="Level 1"/>
5      </Concept>
6    </Entity>
7  </Requirements>
```

**Listing 4.10:** User-defined requirements for all *IfcBuildingElement* entities that are at Level 1

The default *RelatingStructure* for the *IfcWall* is *IfcBuildingStorey*. Nevertheless, if it cannot be assigned to a building storey, the *IfcBuilding* is taken as a spatial container or *IfcSite* when the element is placed on site (buildingSMART International, 2020i). This applies to other elements besides *IfcWall*. Therefore, the quality of the filtered model strongly depends on the quality of the original IFC model, which is to be expected, since all information is taken from this model.

To describe the requirements of a certain entity the elements *Attribute* and *Concept* are used. The *Attribute* element is required to have a *Name* attribute and may also contain a *Value* attribute defining the value of that particular attribute. Both are predefined and must be specified with these XML attribute names. The *Concept* element is somewhat different. It has a *Name* attribute specifying the concept template for which the requirements are given and contains *RuleID* elements. Each *RuleID* element contains attributes, the name of which matches a specific RuleID from the *ConceptTemplate* for a particular attribute, and the given string in the quotation marks is the value of that attribute.

To be able to define the requirements more flexibly, additional special characters were added. These allow not only to define a certain value of an attribute, but create the possibility to specify values in a certain range or to have several values, which all are accepted. For this purpose the [or], [and], [<] and [>] can be added in any string value (Listing 4.9). The latter two can be used by the user to search for all values that lie within a certain range, which can be bigger or smaller than a certain value. The [or] can be placed between values to indicate that the particular attribute may have one of two values. The [and] on the other side defines two values, which both must be true for the attribute. These special characters can be combined for even more flexibility. In Listing 4.9 row 6 specifies a requirement for the thermal transmittance of the walls, which is in the range of 0.1 to 0.5. This can significantly improve the requirement capabilities in some cases.

In addition to the *Name* attribute of the *Entity* element, a further optional *AllAtr* attribute can be defined. It accepts a boolean value which is used to define whether all attributes of a given entity should be added to the filtered IFC. This is helpful, because by default only the explicitly defined attributes of the entities are added in the final IFC file to reduce the information. For example, the result of the requirements defined in Listing 4.10 is a model that contains only the information for the spatial containment of each entity without the property and quantity sets, materials, etc. This is very useful for certain tasks that require only certain information from the entities.

The *Requirements* element has an optional attribute with the name *SurfaceModel*. If the value of this attribute is set to true, the geometry representation of all entity instances is converted to a surface model, regardless of the initial geometry. This is useful because some software applications are only able to represent explicit geometries and the triangulated surface model is the most basic model. Therefore, it is useful to be able to modify the geometry

representation of a IFC model in addition to reducing the semantic information for easier work with the resulting file. In general the conversion is only possible in one direction from implicit to explicit geometry, as described in Chapter 2.1.5. This work is further limited to the ability to convert only into a triangulated surfaces resulting in a *IfcFaceBasedSurfaceModel* that contains many single *IfcFace* elements. Each *IfcFace* is described by 3 points with the corresponding coordinates of the triangle.

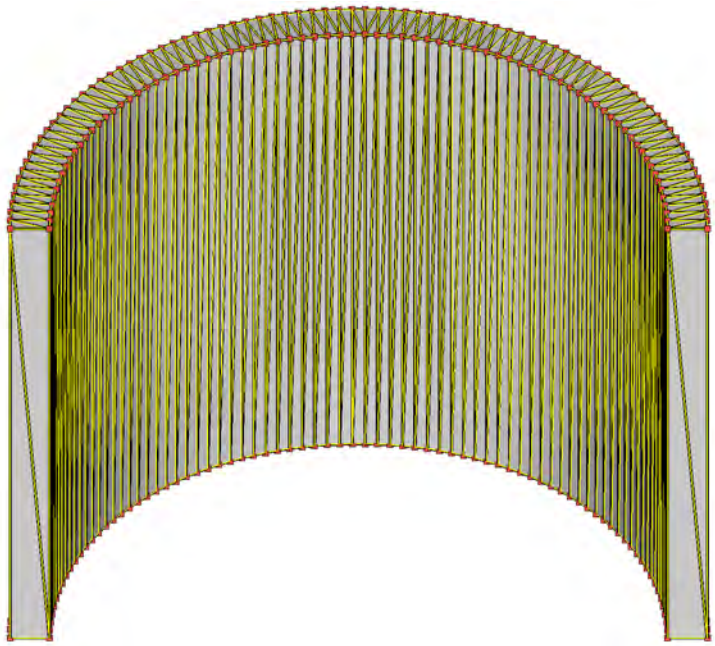


**Figure 4.7:** Curved wall with a triangulated surface geometry displayed in FZK Viewer

The triangulated surface model is easy to interpret from a software application and is therefore used in this thesis as a general geometry representation into which any model can be transformed. However, the biggest challenge of this method is the description of curved surfaces, which cannot be described perfectly, but only approximately. The quality of the approximation depends on the number of triangles used to describe the geometry. This leads to a large number of triangles and the resulting file size is generally much larger compared to a model with implicit geometry. In Figure 4.7 is a wall with a triangulated surface geometry representation, viewed in the FZK Viewer. As shown, each surface of the wall is represented with a number of triangles, with the curved side requiring many triangles. IfcOpenShell (IOS) allows to convert any existing geometry into a triangulated surface using the *tesselate* method. Since a curved geometry is approximated with triangles, the number of triangles and thus the quality of the geometry can be determined as needed. The IOS function *tesselate* therefore offers a parameter to define how fine the triangulation is generated. The smaller values indicate a better approximation. The value of this function argument is left at 1 by default, but can be added as an attribute to the requirements in the future, allowing the user to define the quality of the resulting triangulated surface.

## 4.5 General workflow of the program

After the user requirements are formulated in an XML file, the next step is to use them to filter certain elements from a STEP Physical File. For this purpose the corresponding XML and SPF files must be selected. It is possible to filter either a single or multiple SPFs, each of which has a corresponding user-defined XML requirements assigned to it. If the user specifies multiple IFC files as input, it is important to check whether the elements from each file can be merged into a single file at the end. This is done by comparing the IFC versions of the selected SPFs. If these are not the same, the program is terminated, since only elements from the same IFC version can be merged. In Figure 4.9 is presented a diagram that shows the general workflow of the program and can be used as a reference for the explanation in this section.

The *IfcProject* entity provides important information about the context of all elements contained in the SPF file. This includes the default units required to display the elements properly. A problem that may occur because of the incorrectly defined units could be that some elements are displayed much larger than others. For example, if an IFC has millimeters as its standard units and another uses meters, the elements modeled with the corresponding units will have representations thousands of times smaller or larger. Therefore, the default units must be checked, as this could lead to entities that are disproportionately large. The *IfcProject* also contains important information about the geometric representation context for exchange structures (buildingSMART International, 2020n). This data is presented in an *IfcGeometricRepresentationContext* entity and contains among others information about the project coordinate system and the true north definition. Both are the key to the correct representation of IFC elements with the accurate position and rotation.

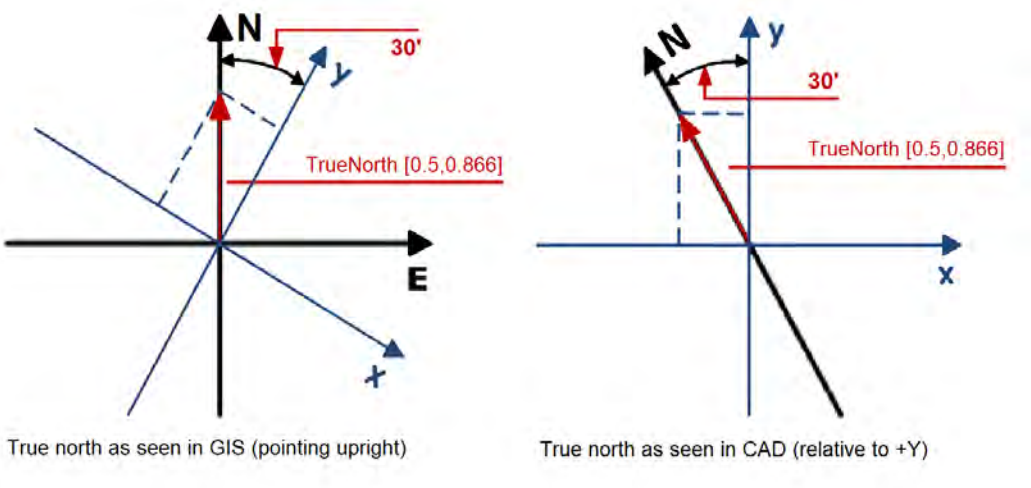



**Figure 4.8:** Definition of the true north direction (buildingSMART International, 2020c)

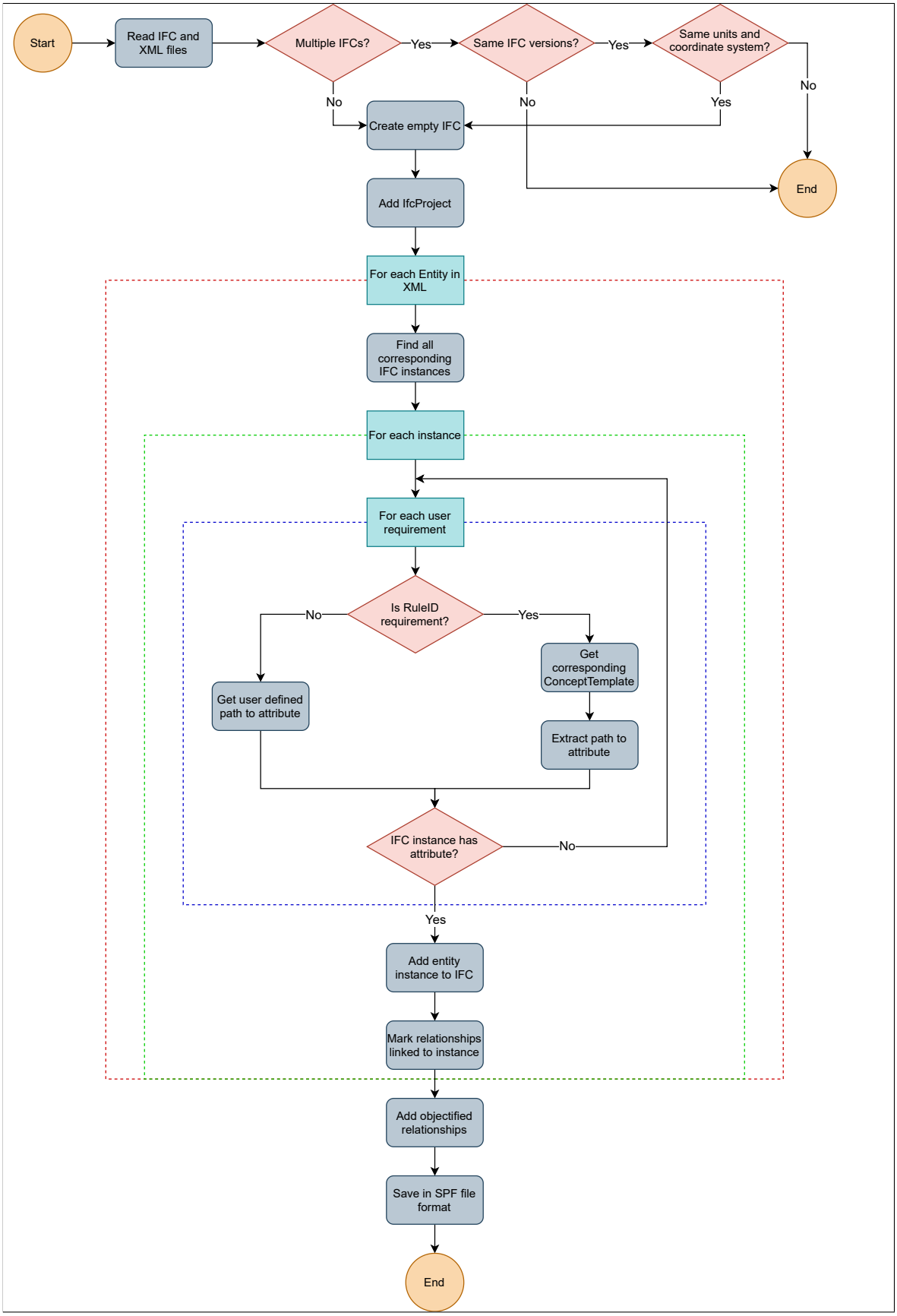



**Figure 4.9:** General workflow of the developed tool

By definition, the use of one instance of the *IfcGeometricRepresentationContext* is mandatory for the representation of the 3D model view and a second instance can optionally be used for the representation of the 2D plan view (buildingSMART International, 2020c). There is often a difference between the north of the 3D model and the geographic north direction, so the *TrueNorth* attribute can be given by a 2-dimensional direction within the xy-plane of the project coordinate system (see Figure 4.8). If the *TrueNorth* is not present, the direction is set to [0.,1.] by default, which means that the positive Y-axis of the project coordinate system is equal to the geographic north direction (buildingSMART International, 2020c). The default units of the project and the data in the *IfcGeometricRepresentationContext* have to be the same in each IFC to continue with the program (Figure 4.9).

After checking the general information from the input SPFs, the next step is to create a new IFC with an empty *DATA* section. This can be done with the *file* method from IfcOpenShell (IOS). The method accepts a string value as an argument that defines the IFC version. The IFC version, e.g. IFC4, is retrieved from the input SPF file so that the resulting output file has the same version at the end. If there are multiple IFC files, it is required for all to have the same version, therefore the resulting SPF has the same version as all input files. The IOS function *file* generates the *HEADER* information for the resulting file together with empty *DATA* section where all filtered information will be stored. Such an empty SPF can be saved in a file with the .ifc extension and contains the data displayed in Listing 4.11 for the IFC4 version.

```
1 ISO−10303−21;
2 HEADER;
3 FILE_DESCRIPTION((’ViewDefinition [CoordinationView]’),’2;1’);
4 FILE_NAME(’’,’2020−11−16T17:50:17’,(),(),’IfcOpenShell 0.6.0b0’,’IfcOpenShell 0.6.0b0’,’’);
5 FILE_SCHEMA((’IFC4’));
6 ENDSEC;
7 DATA;
8 ENDSEC;
9 END−ISO−10303−21;
```

**Listing 4.11:** Empty STEP Physical File created with IfcOpenShell

As already explained, the *IfcProject* contains important information for the representation of the IFC elements. If this data is missing in the STEP Physical File, there could be issues with the size and position of the elements. A problem that has been observed due to missing *IfcProject* information is the incorrect representation of *IfcAlignment* objects in the IFC4x1. In Figure 4.10 are displayed two screenshots from the FZK Viewer of *IfcAlignment* instances. Both represent exactly the same *IfcAlignment* objects, but are displayed differently. The right image (b) shows the correct representation, where the *IfcProject* is present in the SPF. Image (a) in Figure 4.10, on the other hand, shows the *IfcAlignment* instances incorrectly with a significant deviation from the actual geometry. The coordinate system in the FZK

viewer is also displayed disproportionately large compared to the IFC objects because of the undefined default units. This example shows the significance of the *IfcProject* for each SPF. Therefore, after creating the empty IFC file, the next step is to add this entity to it.

**(a)** False *IfcAlignment* representation with missing *IfcProject*

**(b)** Correct *IfcAlignment* representation with *IfcProject*

**Figure 4.10:** Multiple *IfcAlignment* instances displayed with the FZK Viewer

After creating an IFC using the IfcOpenShell *file* method, entities can be added to the file. This can be done with the *create_entity* method of the the *file* class, therefore any file created with IOS has this function. The entity type to be created is passed to the method as a string together with the values of the attributes that this entity holds. Listing 4.12 shows an example of creating a new IFC and adding an *IfcProject* entity to this entity. As shown, the entity type is specified with a string value that corresponds to the entity name. The *GlobalId* attribute of the entity is additionally specified. All direct attributes of an entity can be defined in a similar way, but not the inverse attributes, which refer to the objectified relationships. These must be added additionally and contain the relationship to both the related and relating entities.

```
1 ifc = ifcopenshell.file(schema="IFC4")
2 ifc.create_entity("IfcProject", GlobalId=ifcopenshell.guid.new())
3 ifc.write("/path/new.ifc")
```

**Listing 4.12:** Creating an empty IFC and adding an *IfcProject* with defined *GlobalId* using IfcOpenShell in Python

The IOS *write* method can be used at the end to store the information that has been created in a STEP Physical File. The resulting file contains the sections *HEADER* and *DATA*. The latter contains the information of the newly created entity. The *create_entity* function creates a single line with the corresponding data. In Listing 4.13 the *IfcProject* is the only entity,

since only one entity has been added to this file. The *GlobalId* is specified as a string, while all other attributes are marked with a $ character, which means that they are empty and therefore have no value.

```
1 ...
2 DATA;
3 #1=IFCPROJECT('31voKJKDv4L9sguUguFIne',$,$,$,$,$,$,$,$);
4 ENDSEC;
5 ...
```

**Listing 4.13:** The *DATA* section of the resulting STEP Physical File from the code in Listing 4.12

The *create_entity* method is very useful, and the use of a string value to define the entity type also makes it flexible. However, there is another method to add an existing entity from one IFC file to another. This is done with the IfcOpenShell method *add*. It requires an IFC entity as an attribute. The entity can be retrieved from the input SPF file. Entities of a specific type can be retrieved from an IFC using the IOS *by_type* function. This method is very convenient because it finds all entity instances of a given type. The entity type is again passed to the function as a string. Listing 4.14 shows how this method is used. The function *by_type* returns a list of all instances of this particular type that were found in the IFC model. Therefore, to obtain a specific instance, it is required to access this particular instance in the resulting list. After a certain entity instance is present, it can be added to another IFC (line 5).

```
1 ifc = ifcopenshell.open("/path/existing.ifc")
2 project = ifc.by_type("IfcProject")[0]
3
4 new_ifc = ifcopenshell.file(schema="IFC4")
5 new_ifc.add(project)
6 new_ifc.write("/path/new.ifc")
```

**Listing 4.14:** Retrieving an *IfcProject* from one IFC and adding it to another using IfcOpenShell in Python

The *add* method adds not only the specific IFC entity instance with its attributes, but also the entity instances to which the attributes point. Listing 4.15 shows a part of a SPF to which only a single *IfcProject* entity instance from another IFC has been added. As it can be seen, not only the *IfcProject* was added, but also numerous other entities. All of these contain information related to the *IfcProject*. Line 8 describes the added *IfcProject* together with its attributes. Some of the attributes have string values, while others point to other entities. The *IfcGeometricRepresentationContext*, discussed earlier, is an entity identified with the #10 to which the *RepresentationContexts* attribute of the *IfcProject* refers. The

same applies to the *IfcUnitAssignment* and the *IfcOwnerHistory*. All this data can also be added manually by creating each entity and linking it to the corresponding attribute of other entities. However, since this work is based on already existing SPF files with information that can be easily accessed, this work is based on the *add* method provided by IOS, which significantly reducing the manual creation of entity instances, which is a complex and error-prone process. Nevertheless, the objectified relationships associated with a given entity are not added using this method, as mentioned previously. Therefore, they are added manually, the process of which is explained later in this chapter.

```
1  DATA;
2  ...
3  #6=IFCOWNERHISTORY(#3,#5,$,.NOCHANGE.,$,$,$,1605185851);
4  ...
5  #10=IFCGEOMETRICREPRESENTATIONCONTEXT($,'Model',3,0.01,#8,#9);
6  ...
7  #66=IFCUNITASSIGNMENT((#11,#12,#13,#17,#18,#22,...));
8  #67=IFCPROJECT('1UTGLFNkX8WgcB1e50fxM$',#6,'0001',$,$,'Project Name','
       Project Status',(#10),#66);
9  ENDSEC;
10 ...
```

**Listing 4.15:** Simplified part of the resulting STEP Physical File after adding *IfcProject* from an already existing IFC

After the *IfcProject* is added to the newly created IFC, the next step is to iterate through each *Entity* element specified by the user in the XML file that defines the entity type (Figure 4.9). All entity instances of the defined type are found in the existing *IFC* file using the IOS *by_type* method described above. If there are multiple IFCs, this process is performed for each XML file and the corresponding STEP Physical File. The user-defined requirements for each entity type are checked against each entity instance. For an instance of particular type to be approved and added to the new IFC, it must have all attributes with the values defined by the user.

The user requirements can be defined either with the XML *Attribute* element specifying the full path to a nested attribute or by using the *RuleID* of a certain attribute defined in the mvdXML of the *ConceptTemplate*. The information for the concepts of the IFC entities has been previously downloaded and stored locally for easy access (see Chapter 4.4 for more information). If the user specifies an attribute path, the corresponding attributes of each entity instance are checked against it, and if they exist and have the same value, the respective instance is added to the IFC file. If, on the other hand, a concept with a given *RuleID* is defined, the respective *ConceptTeplate* is accessed and the path to the attribute with the given *RuleID* is extracted from the mvdXML file and used to access the attribute in the IFC

file. Both methods are implemented in Python using a recursive function which searches through all elements in the given path. For example, if the user defines a certain value for a property (see Listing 4.9), the program must check all properties within the property sets. If no such property is found or no property value is equal to the user-defined value, the entity instance is not further examined. It is sufficient that only one of the user criteria is not met by a given entity to discard it as a correctly found item and the program continues with the next entity instance of that type.

IfcOpenShell adds an entity together with the value of its direct attributes as shown previously. This includes the geometry representation and the object placement of each entity that has these. The most common object placement is the local placement as described in Chapter 2.1.6. It defines a relative placement of an entity in relation to another object. However, it depends on the coordinate system defined in the geometric representation context in the *IfcProject*, since the *IfcLocalPlacement* entities are related to each other in a nested structure until the last one is placed within the world coordinate system defined in the *IfcProject*. If the user requires a surface model, the geometry representation of the existing IFC entity is converted to a triangulated surface before it is added to the new IFC (see Chapter 4.4.4). Furthermore, if the attribute *AllAttr* of the XML *Entity* element is provided with a true value by the user, the inverse attribute values are added in addition to the direct attributes of an entity. The inverse attributes point to objectified relationship entities.

As already explained, unlike direct attributes, the inverse attributes of an entity instance are not directly added to the new IFC using the IOS method *add*. The inverse attributes are linked to an objectified relationship entity that defines the relationship between other entities (see Chapter 2.1.4). For example, the *IfcRelContainedInSpatialStructure* objectified relationship is used to assign elements to a certain level of the spatial project structure (buildingSMART International, 2020o). It has an attribute *RelatedElements* that refers to all entity instances in the entire IFC model that are related to this objectified relationship. The attribute *RelatingStructure* points to the spatial structure that contains all *RelatedElements*. All these related elements have an inverse attribute *ContainedInStructure*, which can be used to access the spatial containment of a certain entity. All objectified relationships function in a similar way, having "Related" or "Relating" in their attribute names, which refer to the corresponding entities.

Listing 4.16 introduces a part of a SPFconsisting of an *IfcWall* contained in an *IfcBuildingStorey*. These two entities are connected with the objectified relationship *IfcRelContainedInSpatialStructure*. Neither entity has information about the other, so the objectified relationship is the only connection between them. In this case, the wall is completely independent of the building storey if the relationship is missing. This is important because this data cannot be accessed directly from a specific entity, so the receiving software must correctly read all complex relationships provided in a SPF. The *IfcRelContainedInSpatialStructure* refers to both the *IfcWall* (#207) and the *IfcBuildingStorey* (#151), which are the

*RelatedElements* and the *RelatingStructure* respectively. As explained, it is allowed to have multiple *RelatingElements*, which is often the case. As shown, the *IfcLocalPlacement* of the *IfcWall* (#188) places the entity instance with certain position (#185) relative to the placement of the *IfcBuildingStorey*. This is recognized by the placement of both the wall and the building storey pointing to the *IfcLocalPlacement* on line 10 (#149). This in turn is placed relative to the *IfcCartesianPoint* on line 3 (#6), which is also used to define the world coordinate system in the *IfcGeometricRepresentationContext*. Therefore, all entity placements depend on the coordinate system defined in the *IfcProject*, as explained above.

```
1  DATA;
2  ...
3  #6= IFCCARTESIANPOINT((0.,0.,0.));
4  ...
5  #111= IFCAXIS2PLACEMENT3D(#6,$,$);
6  #112= IFCDIRECTION((6.12303176911189E-17,1.));
7  #114= IFCGEOMETRICREPRESENTATIONCONTEXT($,'Model',3,0.01,#111,#112);
8  ...
9  #148= IFCAXIS2PLACEMENT3D(#6,$,$);
10 #149= IFCLOCALPLACEMENT(#33,#148);
11 #151= IFCBUILDINGSTOREY('0svl7n0uLAchjkoN2YKaIo',#42,'Level 1',$,'Level:8mm
       Head',#149,$,'Level 1',.ELEMENT.,0.);
12 ...
13 #342= IFCRELCONTAINEDINSPATIALSTRUCTURE('3Zu5Bv0LOHrPC10066FoQQ',#42,$,$
       ,(#207),#151);
14 ...
15 #185= IFCCARTESIANPOINT((-6908.90773634549,-206.816459289012,0.));
16 #187= IFCAXIS2PLACEMENT3D(#185,$,$);
17 #188= IFCLOCALPLACEMENT(#149,#187);
18 ...
19 #207= IFCWALL('09h4GnkEf9SO3lQDx1U0PW',#42,'Basic Wall:Generic - 200mm
       :346584',$,'Basic Wall:Generic - 200mm',#188,#202,'346584',.NOTDEFINED.);
20 ...
21 ENDSEC;
22 ...
```

**Listing 4.16:** Simplified part of STEP Physical File showing an objectified relationship and local placement

An objectified relationship can be added to an IFC file in the same way as any entity using the IOS method *add*. However, this should not be performed directly as IfcOpenShell also adds all "Related" and "Relating" entities. For example, if an *IfcWall* is contained in an

*IfcBuildingStorey* and the user requests this wall together with its spatial containment, all other elements contained in this spatial structure element are also added. This will result in elements being added to the resulting IFC file that are not requested by the user. As the purpose of this work is to filter only certain elements, this method cannot be used, since unwanted entities are added automatically. Therefore, after adding a particular entity instance to the new IFC, the *GlobalIds* of the objectified relationships associated with this instance are stored in a Python dictionary. The keys of this dictionary are the *GlobalIds* of the objectified relationship entities, where the value for each key is a list of the *GlobalIds* of the elements that are related to this particular entity. The *GlobalId* is used to uniquely identify an IFC object (see Chapter 2.1.3). IfcOpenShell provides a method *by_id* that finds a specific entity instance from the entire IFC model. After the program has checked all entity instances of the types given by the user and the corresponding entities are added to the newly created IFC, the required objectified relationships are also added. This is done by getting each objectified relationship entity by its *GlobalId*, which is stored in the dictionary explained above, and by changing its "Related" attribute to contain only the entities that are filtered. After only the filtered entities are left, the objectified relationship is added to the new IFC. This automatically adds the entities linked to the "Related" attribute. However, this is necessary because the objectified relationship only connects other entities and does not contain the actual data of the property itself. For example, by adding the entity *IfcRelContainedInSpatialStructure*, the *RelatingStructure*, such as the *IfcSite*, *IfcBuilding* and *IfcBuildingStorey*, is also included in the resulting IFC.

In conclusion, the user defines requirements in an XML file format that specify which elements to filter from one or more IFC files. These requirements are used to define the properties that each entity type should have at the end, which can significantly reduce the complexity and thus the file size of the resulting IFC file. The corresponding entity instances from the input files are checked against the user criteria.This can be seen as a validation process, since the IFC entities must have the attributes and values that are defined by the user. All entity instances that meet the criteria are added to a new IFC together with the required objectified relationships that are added additionally at the end of the process. The ultimate goal is to create a new valid IFC model containing only a part of the initial data. After the required information is added to the new IFC, it is saved in a .ifc file format, so that it can be opened by any program that supports this format. In addition, another IFC file is created in the process, which contains all elements that do not meet the user criteria. This is useful to validate the IFC file and get a result only of the elements that do not have the required information. Such elements may be unsuitable for certain tasks, since they do not provide the necessary data.

## 4.6 Retrieving the IFC models from a CDE with a GUI

### 4.6.1 Overview

A Common Data Environment (CDE) is a BIM environment for storing and managing information in a central repository. With huge amounts of digital data being created and shared during the lifecycle of a project, which is constantly increasing over the years, CDE platforms are becoming increasingly common worldwide. A CDE aims to improve collaboration between project members, reduce errors and avoid duplication of data (McPartland, R., 2016). Project participants retrieve input data from the CDE and in turn store their output data in it (Borrmann *et al.*, 2018). Each user has a specific access level that gives him or her permission to interact with certain information. There are a number of CDE platforms, with Autodesk BIM360 being one of the most popular. The BIM360 Docs can be used to manage all types of documents and is used in this paper to demonstrate the workflow of the developed tool.

BIM360 provides a web application that project participants can use in a web browser. However, to access and use the data stored in the CDE with the custom tool developed for this work, the Autodesk Forge APIs and services are used (Autodesk Forge, 2020a). An API allows to programmatically connect with a separate software component or resource and thus provides the possibility to create interaction between multiple applications. The BIM360 platform can be accessed via RESTful APIs by using HTTP requests to manage the data. The data can be uploaded, modified and retrieved using GET, POST, DELETE, etc. request methods. These requests can be executed using the Python library *requests* and return a response containing the required information (see Chapter 4.3.3).

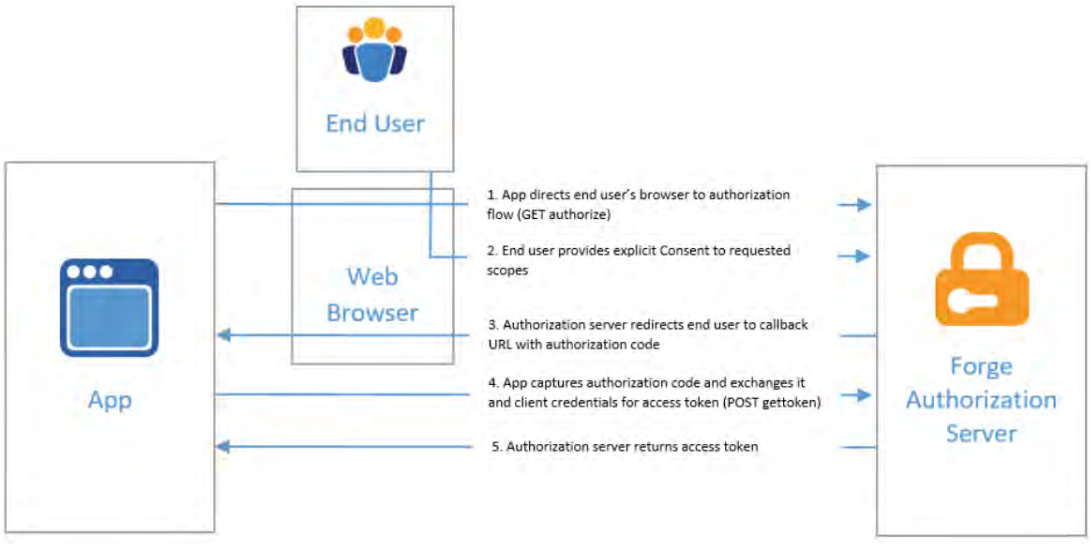



**Figure 4.11:** Authentication process for retrieving a 3-Legged Token (Autodesk Forge, 2020b)

To retrieve the information stored in BIM360 Docs to which a particular user has access via the Forge APIs, a 3-legged token must be obtained, which is used to verify the user's identity. The process for this is illustrated in Figure 4.11. The first step is to open the web browser and redirect the user to the authorization flow. The user must first log in using his or her credentials. After a successful login, the user will be asked to give the explicit consent to the requested scopes, which are explained in more detail in the following section. The next step is to receive the authorization code provided in a callback URL to which the user is redirected. In the final step this code is exchanged for an access token that can be used in the developed tool to access the CDE information. This complex authorization process is required to validate the identity of the user and to ensure the security of the data stored in the BIM360 platform.

### 4.6.2 Implementation with a GUI

A Graphical User Interface (GUI) is essential for any software application to allow the user to interact with it without the need for programming knowledge. Therefore a simple GUI was developed for this work, which allows the user to visually navigate and interact with the provided information. After the program has been started, the user can choose between two options: get files from BIM360 Docs or get local files. The second option allows the user to select IFC files that are stored locally on the device. For each chosen IFC file, a corresponding XML file must also be selected, in which the user requirements are defined. On the other hand, if the user chooses to obtain the IFC files from BIM360, the program automatically opens the default web browser and requests the user's credentials to log in. After a successful login, the scopes requested by the program must be accepted by the user (explained in the previous section).

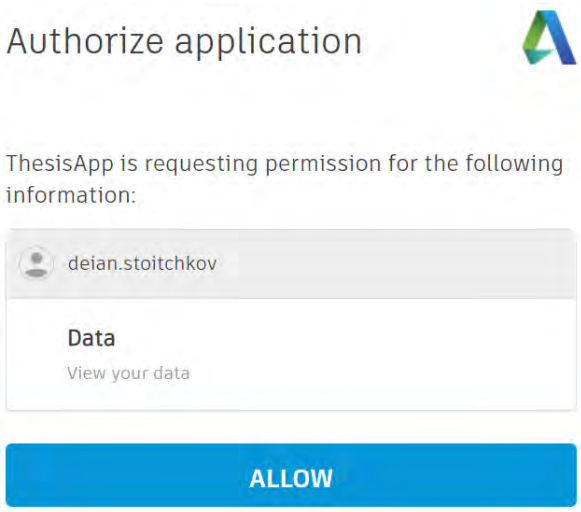



**Figure 4.12:** Authorization flow in the web browser

Since the developed tool only requires access to the IFC files from the CDE without modifying them, the program only requests permission to read the data (Figure 4.12). The required permissions are given as a string to the HTTP request with the available values for the data being: *data:read*, *data:write*, *data:create*, *data:search*.

After the user has allowed the program to view the data from BIM360, the developed GUI can be used for visual interaction with the data in the CDE (Figure 4.13). The GUI is created using the *Tkinter* Python library, which is the most commonly used library for this purpose as it provides a fast and easy way to create GUI applications. BIM360 consists of hubs containing projects, which in turn have folder structures to each of which user permissions are assigned. Furthermore, the folder structures for each project are divided into *Plans* and *Project Files*. Files stored in BIM360 can be downloaded via the API only if they are in the *Project Files* directory. Therefore, all IFC files must be saved in this folder in order to access them. For simplicity, the location of the files must always be in the root *Project Files* folder and the nested folder structure cannot be accessed.

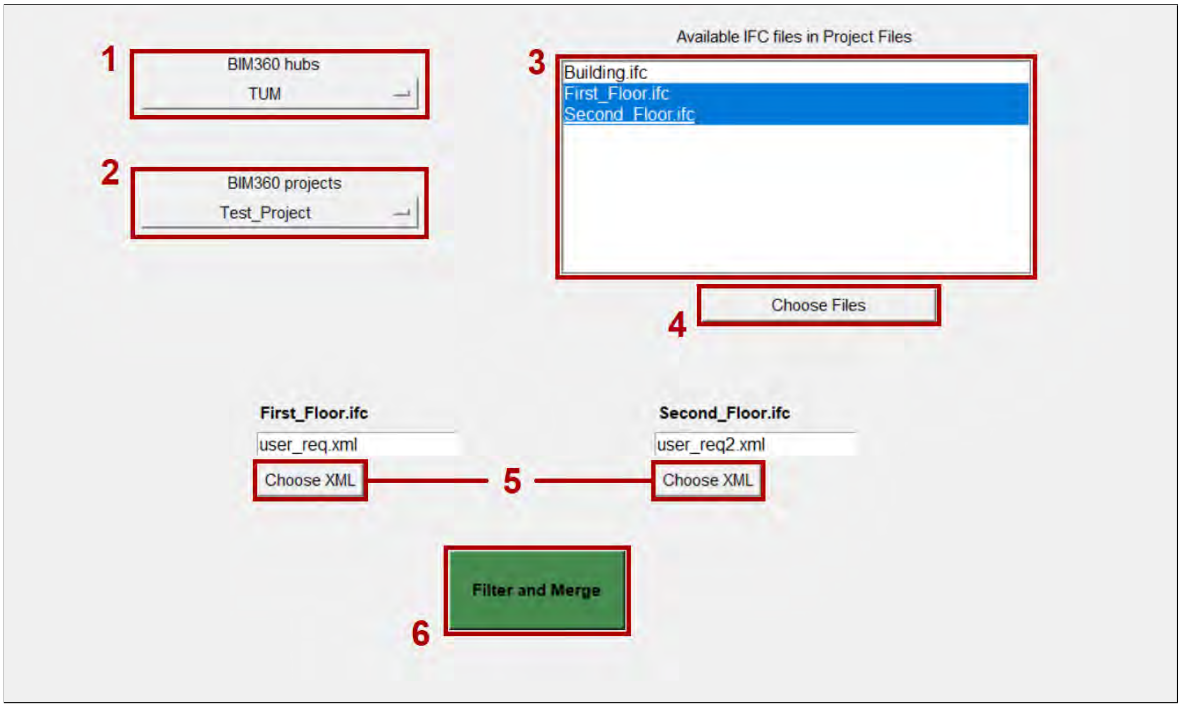



**Figure 4.13:** GUI for accessing files from BIM360

Figure 4.13 displays the Graphical User Interface for the workflow of the program for getting the IFC files from the BIM360 Docs. The first step is to select the BIM360 hub from the drop-down menu (1). This menu automatically shows all hubs available for this user that have been retrieved with the Forge APIs. Once the hub has been selected, the project containing the files can be chosen, which again is done from a drop-down menu with all possible values (2). Next, all available IFC files in the root *Project Files* folder are displayed (3). The user

can select one or more files from the list and click the *Choose Files* button (4). This will automatically download the required information and prepare it for the filtering and merging process. For each selected IFC file, a corresponding XML file containing the user-defined requirements must be chosen (5). If only one file is selected, the program filters it and creates a new valid IFC model, which is stored locally on the device. On the other hand, if more than one file is selected, the program filters and merges them together into a single IFC file (6). Using the Forge APIs to access the data from BIM360 is an essential component as it allows the user to retrieve the data and transform it in a single program without having to manually download and using it in another software.

# Chapter 5

# Case study

## 5.1 Overview

The ability of the developed tool to handle typical use case scenarios is evaluated using two different IFC files. For each IFC, a corresponding XML file was prepared to create specific user-defined requirements for filtering particular elements. The process of the program is the same in each case, retrieving the input SPF files, automatically creating a new valid IFC and adding the requested elements along with the specific properties assigned by the user.

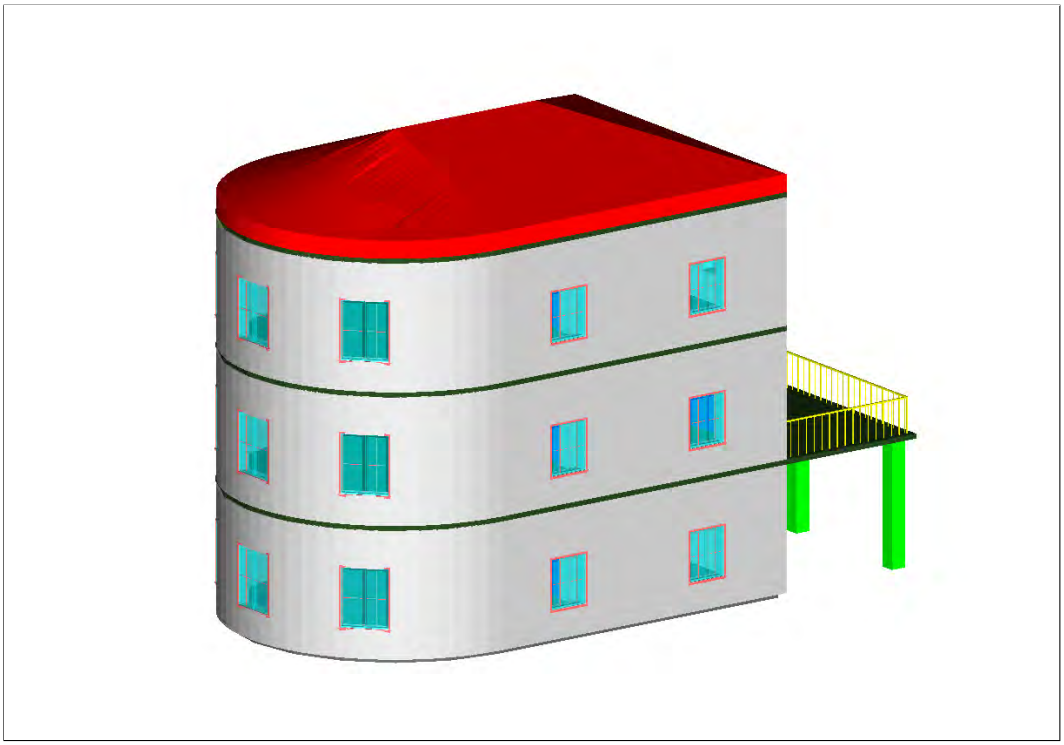
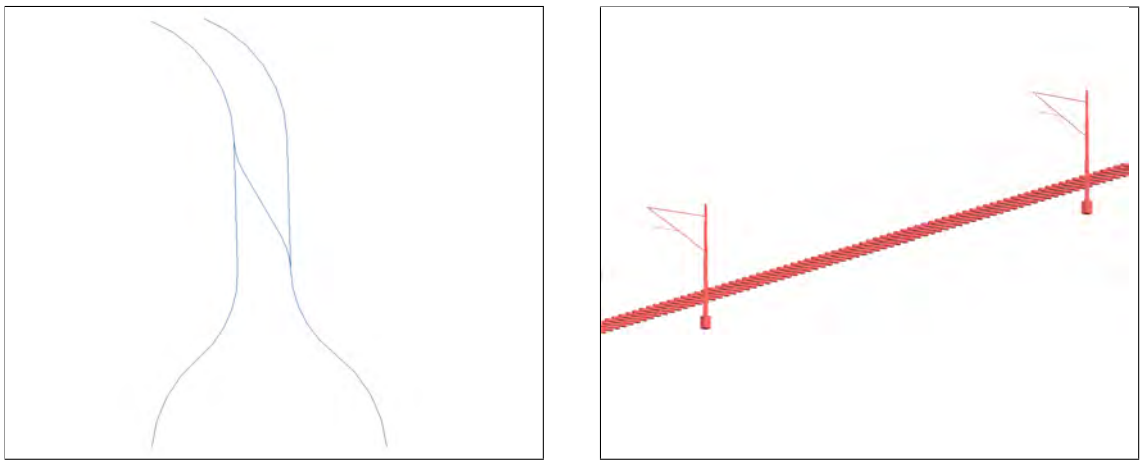


**Figure 5.1:** Used example IFC model of a 3 storey building

The workflow of the developed program is described in detail in chapter 4.5 and in this chapter only the initial models, the XML requirements and the final result are shown. The FZK Viewer is used for all IFC models to open and visually display the initial and resulting elements. The FZK Viewer has established itself as the independent open source viewer for IFC files and is therefore the choice for this work to correctly represent both the geometry and the semantics of the IFC entities.

The first IFC model used to demonstrate the functionalities of the developed tool is a model of simple three-story building. It was created using Autodesk Revit 2021 and exported to IFC4 using the *Reference View*, which is an official Model View Definition (MVD) from buildingSMART International (see Chapter 2.3). The IFC model is a basic architectural model and contains walls, slabs, columns, windows, railings and a roof (Figure 5.1). The program developed for this work can generally be used for much more complex models. However, for a better visual representation of the elements, a simple model was chosen, which is sufficient to show the capabilities of the tool.

The second model is an IFC4x1 model of a long railroad line consisting of more than 5000 individual elements representing the railroad sleepers (Figure 5.2). In addition, the catenary masts are arranged in a certain distance from each other along the line (Figure 5.2 (b)). The elements are placed linearly along the *IfcAlignment* entity instances. Such an IFC model often turns out to be very large and there might be difficulties in opening it. The FZK Viewer requires more than 20 minutes to open the corresponding SPF. Therefore, it is highly beneficial to have the possibility to filter and extract certain elements, thus reducing the overall size of the file and making the collaboration between project participants more convenient. The next section provides examples of user requirements and the resulting models.



**(a)** Full track **(b)** Zoomed in part of the model

**Figure 5.2:** Used example IFC model of a railway line

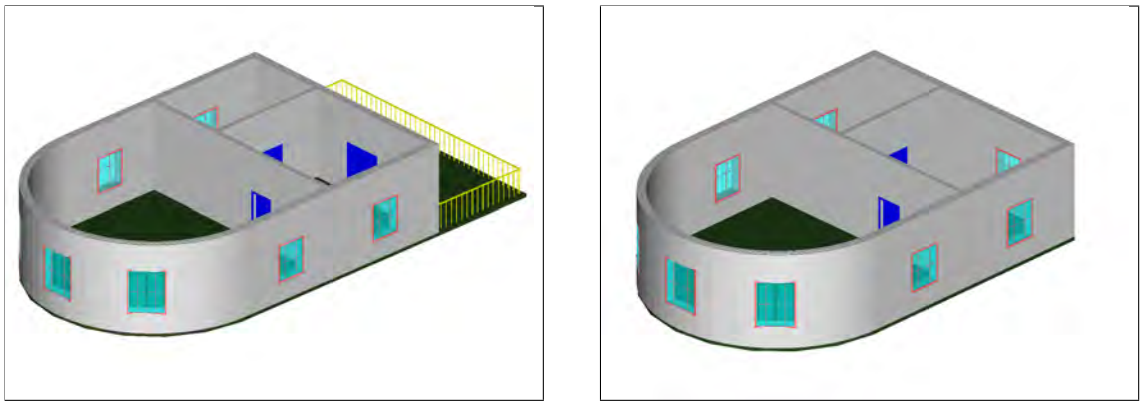
## 5.2 Filtering and merging using user-defined requirements

First, the model displayed in Figure 5.1 is filtered to extract only the elements of a certain building storey. For this purpose, the XML file with the criteria defined in Listing 5.1 is used. It is requested by the user to retrieve all *IfcBuildingElement* entity instances that can be found in the spatial element with name "Level 2". This is done using the *RuleID* of the specific attribute from the corresponding *ConceptTemplate* (see Chapter 4.4). Therefore the program finds all building elements contained in the requested building storey and adds them to a new IFC model. However, this only works if the elements are actually contained in a building storey and not the building itself or the site (see Chapter 4.4.4). Furthermore, the attribute *AllAttr* of the *Entity* element is set to true. As a result, all attributes including the inverse attributes together with the objectified relationships of the original entity instances are also added to the resulting model, such as the associated materials, property sets, etc.

```
<Requirements SurfaceModel="False">
  <Entity Type="IfcBuildingElement" AllAttr="True">
    <Concept Name= "Spatial Containment">
      <RuleID SpatialElementName="Level 2"/>
    </Concept>
  </Entity>
</Requirements>
```

**Listing 5.1:** User-defined requirements for filtering *IfcBuildingElement* entity instances

The developed tool can be used several times for the same input IFC model with different filter criteria, whereby each time a new valid STEP Physical File (SPF) is created, which contains only the requested elements. In Figure 5.3 2 IFC models are displayed. Both contain elements extracted from the model in Figure 5.1. The first one (a) contains all *IfcBuildingElement* entities that are placed on the second floor, while the second (b) contains the same types of elements but those located on the third floor of the building. As it is displayed, all building elements, including the walls, the doors, the windows etc., are present and are correctly represented in the newly created SPFs. Both IFC models are the result of using the developed tool for the same initial model with the requirements in Listing 5.1, the only difference being the value of the *SpatialElementName*. After the filtering process, the resulting IFC files are stored in the BIM360 Docs platform, from where they can be automatically accessed as described in chapter 4.6.2. The user can connect to the BIM360 account and automatically retrieve the IFC files using the developed Graphical User Interface (GUI). The user can select the hub and the corresponding project from the BIM360 in which the IFC files are stored. The available IFC models within the chosen project are presented to the user directly within the GUI for filtering and merging (Figure 4.13).



**(a)** Elements contained in the spatial element with name "Level 2"

**(b)** Elements contained in the spatial element with name "Level 3"

**Figure 5.3:** The resulting models after using the tool with the requirements in Listing 5.1

```
<Requirements SurfaceModel="False">
  <Entity Type="IfcWall">
    <Concept Name="Property Sets for Objects" >
      <RuleID PropertyName="ThermalTransmittance" Value="[<]0.2"/>
      <RuleID PropertyName="LoadBearing" Value="True" />
    </Concept>
  </Entity>
  <Entity Type="IfcSlab"/>
</Requirements>
```

**Listing 5.2:** User-defined requirements for retrieving the load-bearing walls and the slabs

The models depicted in Figure 5.3 are further filtered and merged together to demonstrate the capabilities of the developed tool. Both models are stored in the BIM360 and automatically retrieved with the help of the Forge APIs as described in Chapter 4.6. Only the walls that are load-bearing and have a thermal transmittance below 0.2 are requested by the user. In addition, all slabs are also to be included in the final IFC. These requirements are presented in Listing 5.2. As shown, the user criteria for the *IfcWall* entity instances are defined using the *ConceptTemplate* for the property sets and the individual properties are accessed with the corresponding *RuleIDs*. The XML element *Entity* for the *IfcSlab* entities has no sub-elements, which means that all slabs from the original model are added to the resulting SPF. The outcome of this is displayed in Figure 5.4. Elements from both original models are contained in the resulting IFC file. However, only the entities requested by the user are included. Therefore, there are no windows, doors and railings, but only the load-bearing walls and the slabs. All elements are represented correctly with an accurate position. Furthermore, all entities have only the properties defined in the user requirements. The result is a model

with reduced information that is easier to handle for further collaboration and also faster to interpret by many applications. Multiple IFC models, stored in BIM360 Docs, are merged into a single final model that contains only the requested data.

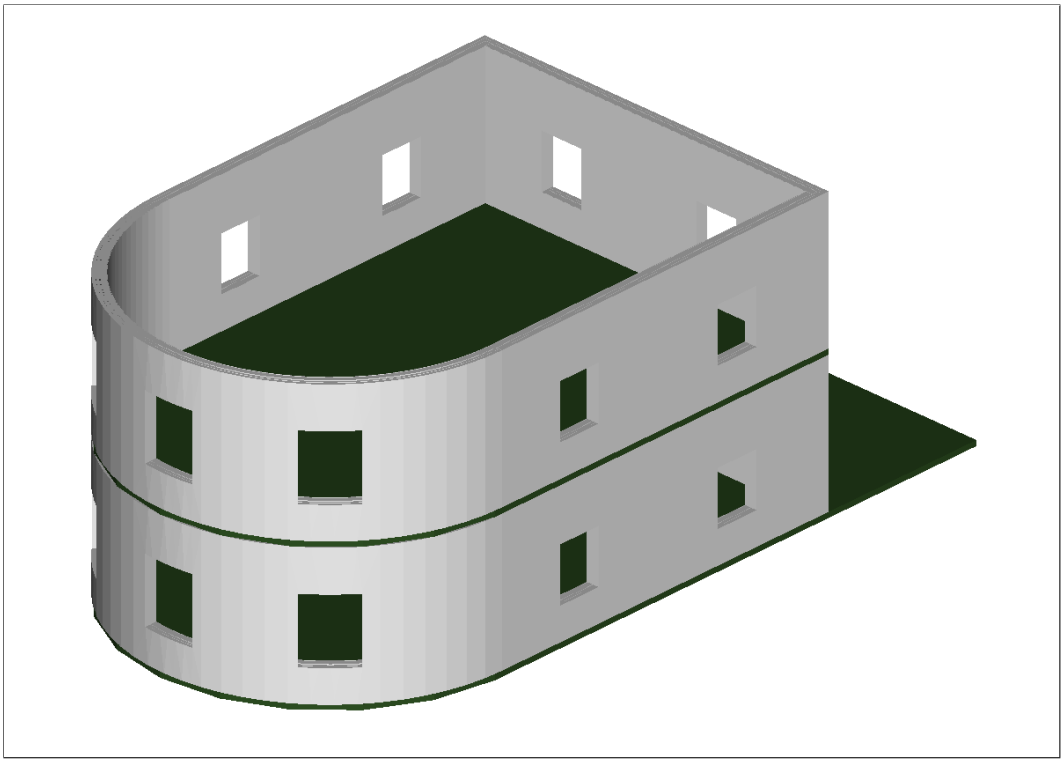


**Figure 5.4:** Resulting model after filtering and merging the models depicted in Figure in 5.3 with the requirements in Listing 5.2

Next, the IFC model of the railway line shown in Figure 5.2 is filtered. For this purpose, the requirements presented in Listing 5.3 are used. All of the railroad sleepers are represented as *IfcBuildingElementProxy* entity instances positioned with an *IfcLinearPlacement*. The attribute *DistanceAlong* shows the distance along the basis curve, in this case an *IfcAlignment* curve (buildingSMART International, 2020p). The requirements in Listing 5.3 specify the railroad sleepers and catenary masts that are located between 300 and 500 kilometers along the curve (line 5). In this case, however, no *ConceptTemplate* can be used to access the attribute and both the general type of the IFC entity describing the elements and the object placement must be known to the user to define the correct requirements. This requires a good understanding of the IFC schema and become quite complex for some attributes, but buildingSMART International provides good documentation defining the different entities and attributes as described in Chapter 2. The *IfcAlignment* entity instances are also added to the resulting file as they are also requested by the user in the XML file (line 9). As previously explained, the corresponding elements are filtered and added to a new valid SPF that is stored locally so that it can be easily further used.

```
1  <Requirements SurfaceModel="True">
2    <Entity Type="IfcBuildingElementProxy">
3      <Attribute Name="ObjectPlacement">
4        <Attribute Name="Distance">
5          <Attribute Name="DistanceAlong" Value="[>]300[and][<]500" />
6        </Attribute>
7      </Attribute>
8    </Entity>
9    <Entity Type="IfcAlignment"/>
10 </Requirements>
```

**Listing 5.3:** User-defined requirements for retrieving elements in a specific distance range along a curve

The *SurfaceModel* attribute of the *Requirements* element is set to true (Listing 5.3 line 1). This converts all geometry representations of the entities into a surface model as described in chapter 4.4.4. This results in a model that is generally larger in than a model with an implicit geometry like the Constructive Solid Geometry (CSG), but is easier to interpret and more commonly accepted by software applications. As already explained, the FZK Viewer requires more than 20 minutes to open the complete IFC model shown in Figure 5.2. Only by converting the geometry of all elements into a surface geometry, the time to open the same data is significantly reduced to less than one minute. In combination with filtering specific elements, this makes the final IFC more convenient for future use. The IFC model resulting from the requirements in Listing 5.3 is displayed in Figure 5.5. As shown, only a part of the elements along the *IfcAlignment* are contained in the new model corresponding to the filtered elements between the 300 and 500 kilometers along the curve.

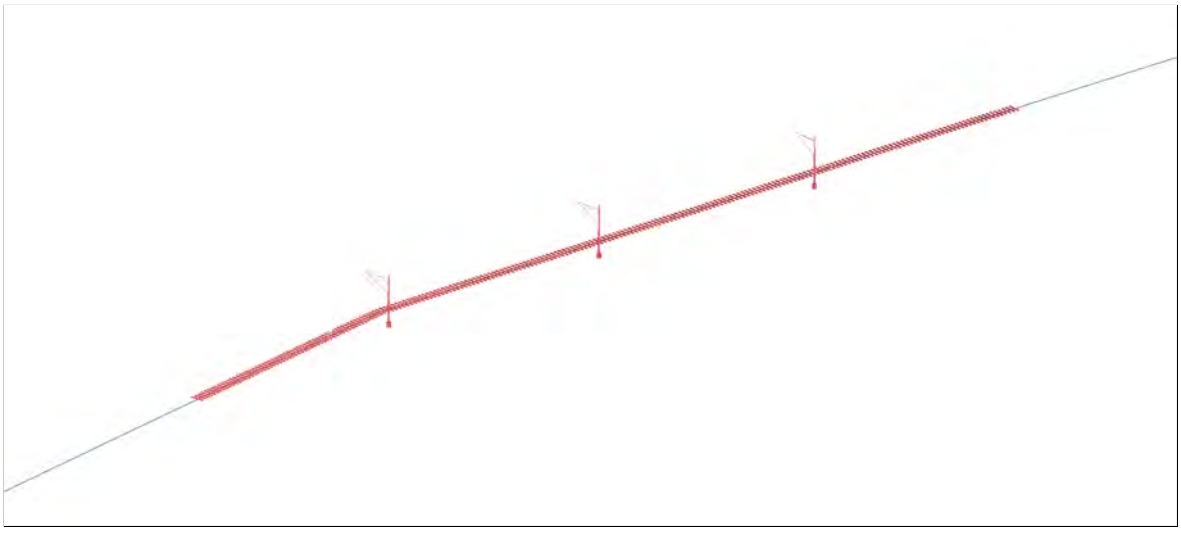


**Figure 5.5:** Resulting filtered model of the railway line

# Chapter 6

# Discussion

## 6.1 Summary

The tool developed in this work is capable of successfully filtering existing IFC models and creating new models that contain only the information requested by the user. The user can create requirements in an XML format that is both human and machine readable. These requirements are used to define certain entity types and their attributes that must be present in the original STEP Physical File (SPF) in order to be added to the resulting model. Due to the complex structure of the IFC schema, which uses objectified relationships to link entities with each other, access to certain properties of an IFC entity can become complicated. As a result, the user needs to have a thorough understanding of the IFC schema in order to define the correct attribute path that needs to be accessed. Therefore, the official data provided by buildingSMART International is used to reduce the complexity of this process. For this purpose, the *ConceptTemplates* available for the different IFC units are automatically accessed and downloaded, so that the information contained in them can be used within the developed application to validate the user criteria. The *RuleIDs*, which some attributes have been given in the corresponding *ConceptTemplate*, can be used by the user to define the requirements in a more simplified way.

An IFC model can not only be filtered with the developed tool, but the elements from multiple models can also be merged into a single IFC file. The models stored in a Common Data Environment (CDE) - Autodesk BIM360 Docs is used in this work - can be automatically accessed through the developed Graphical User Interface (GUI). This is a significant part of the program, as it allows the user to retrieve and transform data that is not locally available in a single stand-alone application designed specifically for this purpose. The geometry representation of the resulting IFC entity instances can also be converted independently of the initial geometry into a triangulated surface model that is easy to interpret by software applications and therefore is widely accepted. Therefore, the developed application allows

filtering and merging of multiple IFC models with user-defined requirements as well as transformation of the geometry representation of the resulting model. This is achieved using the Python programming language. The IfcOpenShell (IOS) library is used to work with the SPFs, which provides everything required to access and modify existing IFC models and to create new models. The Autodesk Forge APIs and services are also used to automatically retrieve the data from BIM360 within the developed tool.

## 6.2 Limitations

The developed tool can be used for general custom requirements to create new valid IFC files that contain a part of the information found in the original models. However, there are some limitations associated with using the tool. In general, the user needs to have a good understanding of the IFC schema, which is a complex subject at first if there is no prior knowledge. The goal of this thesis is to reduce the required experience with the Industry Foundation Classes (IFC) by providing simplified access to the attributes of entities using the *RuleIDs* defined in the *ConceptTemplates*, whose information is easily accessible to everyone, since it is officially provided by buildingSMART International. However, there are only a limited number of attributes that can be accessed with *RuleIDs*, which means that the full path to particular attributes of an entity still has to be defined manually by the user. Since this work is aimed at filtering and merging existing IFC files, another critical factor is the quality of the entered IFC models and the user's knowledge of how the elements are exported. In addition, modeling software applications are always certified for the import or export of IFC models against a specific Model View Definition (MVD) that contains only a portion of the original model data, so there is some data loss when using IFC models. The geometry representation of the resulting IFC file can be converted into a triangulation surface model. However, it is important to note that in general it is only possible to correctly convert implicit into explicit geometry. This work is further limited to the ability to convert any geometry only into the explicit triangulated surface model. Therefore, either the original geometry representation of the IFC model is left or all element geometries are converted into a triangulated surface model. It is also allowed to merge IFC files only if they all share the same version and have the same default units and coordinate system.

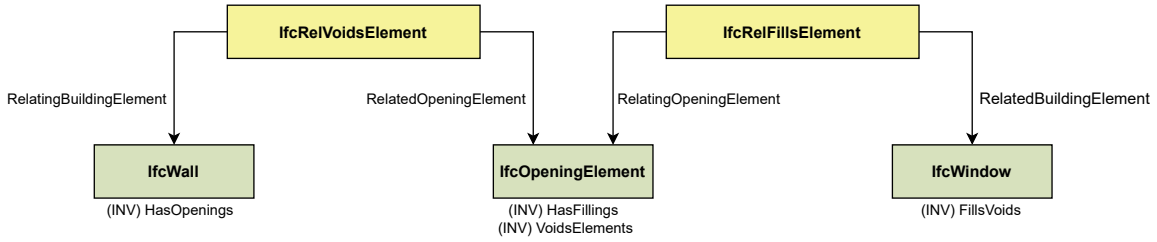



**Figure 6.1:** Wall-Opening-Window relationship

The relationship between the entities is an essential part of any IFC model. However, when merging multiple files, it is not possible to establish relationships between the entities from different IFC files, as these cannot be described by the user. The geometry and position of the entities are still correctly represented relative to each other, but the semantic relationship is missing. Therefore, it is not possible to create requirements, for example to retrieve the walls from one IFC model and the windows from another. These two elements are connected by a middle entity that defines the opening in the wall (Figure 6.1). Both the wall and the window are connected to this middle element by an objectified relationship, the IfcRelVoidsElement and the IfcRelFillsElement respectively.

## 6.3 Conclusion and future outlook

The developed prototype shows promising results in filtering and merging IFC models. A user can define requirements in an XML format to retrieve specific elements from a model. However, this is only a prototype and as explained in the previous section, it is associated with certain limitations. To facilitate the process of requirements creation by the user, the *RuleIDs* of attributes can be used. There are certain predefined *ConceptTemplates* that have been officially created by buildingSMART International. These describe reusable concepts that can be applied to a specific IFC entity. Some attributes in the *ConceptTemplates*, which are described in a mvdXML format, are given a *RuleID*. However, not all attributes have a *RuleID*, which restricts the choice for the user. Furthermore, there is a limited number of *ConceptTemplates* that describe only a part of the possible properties of an entity that can be accessed. Therefore the user still has to define the full path to certain attributes manually, which requires a deep understanding of the IFC schema and is a time-consuming task. A possible solution for this would be that bSI provides a more thorough *ConceptTemplate* definitions with available *RuleIDs* for most of the available attributes that can be accessed for a given entity.

This thesis provides an approach for merging multiple IFC models into a single valid STEP Physical File. The original files can be stored in a Common Data Environment (CDE) and automatically retrieved. For this purpose the BIM360 Docs platform is used to demonstrate the process. However, this requires the use of the Autodesk Forge APIs and services to connect to the CDE and automatically retrieve the data so that the user can access it directly in the developed tool. This is a complex process and in the future the functionality of filtering and merging IFC models could be built-in directly into the CDE platforms to make it easier for users to access and use. In general this is possible, but the biggest problem is how the user defines the required information from a particular model. This always requires at least some understanding of the IFC schema, which is a complex issue. The process could be simplified by a GUI designed specifically for this purpose, which allows the user to select the desired information in a user-friendly way.

# Bibliography

Adachi, Y. (2003). Overview of Partial Model Query Language. In proceedings of the 10th ISPE International Conference on Concurrent Engineering (ISPE CE 2003), 549-555.

Autodesk Forge (2020a). A cloud-based developer platform from Autodesk. https://forge.autodesk.com/, accessed on 2020-11-25.

Autodesk Forge (2020b). Get a 3-Legged Token with Authorization Code Grant. https://forge.autodesk.com/en/docs/oauth/v2/tutorials/get-3-legged-token/, accessed on 2020-11-25.

Autodesk, Inc. (2018). Revit IFC manual. Detailed instructons for handling IFC files. https://damassets.autodesk.net/content/dam/autodesk/draftr/2528/180213_IFC_Handbuch.pdf, accessed on 2020-10-30.

Azhar, Salman. (2011). Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry. Leadership and management in engineering, 11(3), 241-252.

Baldwin, M. (2019). Der BIM-Manager: Praktische Anleitung für das BIM-Projektmanagement. Beuth Verlag.

Baumgärtel, K., Pirnbaum, S., Pruvost, H., & Scherer, R. J. (2016). Automatic BIM filtering using Model View Definitions. In CIB W78 conference, Brisbane, Australia.

Bew, M., & Richards, M. (2011). BIM maturity model, strategy paper for the government construction client group. London: Department of Business, Innovation and Skills. https://www.cdbb.cam.ac.uk/Resources/ResoucePublications/BISBIMstrategyReport.pdf, accessed on 2020-10-22.

BIM Supporters. The IFC Schema basics. https://app.bimsupporters.com/courses/ifc/lessons/the-ifc-schema-basics/, accessed on 2020-10-18.

Borrmann, André; König, Markus; Koch, Christian; Beetz, Jakob (2018) Building Information Modeling: Technology Foundations and Industry Practice. Cham: Springer International Publishing.

Buchele, Suzanne & Crawford, Richard. (2004). Three-dimensional halfspace constructive solid geometry tree construction from implicit boundary representations. Computer-Aided Design. 36. 1063-1073. 10.1016/j.cad.2004.01.006.

buildingSMART GitHub (2020). IfcDoc. https://github.com/buildingSMART/ifcdoc, accessed on 2020-11-09.

buildingSMART International (2020a). Introduction. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/introduction.htm, accessed on 2020-10-15.

buildingSMART International (2020b). IfcObject. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifckernel/lexical/ifcobject.htm, accessed on 2020-10-16.

buildingSMART International (2020c). IfcGeometricRepresentationItem. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometryresource/lexical/ifcgeometricrepresentationitem.htm, accessed on 2020-10-18.

buildingSMART International (2020d). IfcObjectPlacement. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifcobjectplacement.htm, accessed on 2020-10-19.

buildingSMART International (2020e). IfcLocalPlacement. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifclocalplacement.htm, accessed on 2020-10-19.

buildingSMART International (2020f). Model View Definition (MVD) - An Introduction. https://technical.buildingsmart.org/standards/ifc/mvd/, accessed on 2020-10-20.

buildingSMART International (2020g). MVD Database. https://technical.buildingsmart.org/standards/ifc/mvd/mvd-database/, accessed on 2020-10-20.

buildingSMART International (2020h). Index of /IFC/RELEASE. https://standards.buildingsmart.org/IFC/RELEASE/, accessed on 2020-11-07.

buildingSMART International (2020i). IfcWall. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcsharedbldgelements/lexical/ifcwall.htm, accessed on 2020-10-27.

buildingSMART International (2020j). IfcRoot. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifckernel/lexical/ifcroot.htm, accessed on 2020-10-13.

buildingSMART International (2020k). IfcDoc. https://www.buildingsmart.org/standards/groups/ifcdoc/, accessed on 2020-10-31.

buildingSMART International (2020l). Material Layer Set. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/templates/material-layer-set.htm, accessed on 2020-11-10.

buildingSMART International (2020m). Material. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/templates/material.htm, accessed on 2020-11-11.

buildingSMART International (2020n). IfcProject. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifckernel/lexical/ifcproject.htm, accessed on 2020-11-16.

buildingSMART International (2020o). IfcRelContainedInSpatialStructure. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcproductextension/lexical/ifcrelcontainedinspatialstructure.htm, accessed on 2020-11-17.

buildingSMART International (2020p). IfcDistanceExpression. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometryresource/lexical/ifcdistanceexpression.htm, accessed on 2020-11-29.

Chipman, Tim; Liebich, Thomas; Weise, Matthias (2016). mvdXML specification 1.1. Specification of a standardized format to define and exchange Model View Definitions with Exchange Requirements and Validation Rules.

Daum, S., Borrmann, A., Langenhan, C., & Petzold, F. (2014). Automated generation of building fingerprints using a spatio-semantic query language for building information models. eWork and eBusiness in Architecture, Engineering and Construction: ECPPM, 2014, 87.

Daum, S., & Borrmann, A. (2014). Processing of topological BIM queries using boundary representation based methods. Advanced Engineering Informatics, 28(4), 272-286.

Gergana Popgavrilova (2020). Assuring building information quality for building analytics by translating use cases of BIM@SRE standard into the MVD format. Master's thesis. TUM Department of Civil, Geo and Environmental Engineering. Chair of Computational Modeling and Simulation.

Hietanen, J., & Final, S. (2006). IFC model view definition format. International Alliance for Interoperability, 1-29.

IfcOpenShell (2020a). The open source ifc toolkit and geometry engine. http://ifcopenshell.org/, accessed on 2020-11-02.

IfcOpenShell (2020b). GitHub repository. https://github.com/IfcOpenShell/IfcOpenShell, accessed on 2020-11-02.

IfcOpenShell (2020c). IfcOpenShell-python. http://ifcopenshell.org/python, accessed on 2020-11-03.

IFC++ (2020). https://ifcquery.com/, accessed on 2020-11-02.

Karlsruhe Institute of Technology (2020). Institute for Automation and Applied Informatics (IAI). FZKViewer. https://www.iai.kit.edu/english/1648.php, accessed on 2020-11-04.

Karl-Heinz Häfele, Andreas Geiger, Thomas Liebich (2008). Implementation Guide for IFC Header Section. Version 1.0.2

Liebich, T. (2009). IFC 2x Edition 3 Model Implementation Guide. Technical report, buildingSMART International.

Mazairac, W., & Beetz, J. (2013). BIMQL–An open query language for building information models. Advanced Engineering Informatics, 27(4), 444-456.

McPartland, Richard (2016). What is the Common Data Environment (CDE)? NBS. https://www.thenbs.com/knowledge/what-is-the-common-data-environment-cde, accessed on 2020-11-25.

MDN web docs (2020). HTTP request methods. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods, accessed on 2020-11-06.

NBS National BIM Report 2019. https://www.thenbs.com/knowledge/national-bim-report-2019, accessed on 2020-10-29.

Preidel, C., Daum, S., & Borrmann, A. (2017). Data retrieval from building information models based on visual programming. Visualization in Engineering, 5(1), 1-14.

PyCharm (2020). The Python IDE for Professional Developers. https://www.jetbrains.com/pycharm/, accessed on 2020-11-03.

Shapiro, V., & Vossler, D. L. (1993). Separation for boundary to CSG conversion. ACM Transactions on Graphics (TOG), 12(1), 35-55.

Solibri (2020). A Nemetschek company. https://www.solibri.com/, accessed on 2020-11-05.

Stack Overflow 2020 Developer Survey (2020). Most wanted languages. https://insights.stackoverflow.com/survey/2020, accessed on 2020-11-03.

Tauscher, E., Bargstädt, H. J., & Smarsly, K. (2016, July). Generic BIM queries based on the IFC object model using graph theory. In The 16th International Conference on Computing in Civil and Building Engineering.

Tobiáš, Pavel. (2015). An Investigation into the Possibilities of BIM and GIS Cooperation and Utilization of GIS in the BIM Process. Geoinformatics FCE CTU. 14. 65. 10.14311/gi.14.1.5..

WebHarvy (2020). What is Web Scraping? https://www.webharvy.com/articles/what-is-web-scraping.html, accessed on 2020-11-05.

Weise, M., Liebich, T., Nisbet, N., & Benghi, C. (2017). IFC model checking based on mvdXML 1.1. eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2016, 19-26

Weise, M., Katranuschkov, P., & Scherer, R. J. (2003). Generalised model subset definition schema. CIB report, 284, 440.

Windisch, R., Katranuschkov, P., & Scherer, R. J. (2012, July). A generic filter framework for consistent generation of BIM-based model views. In Proceedings of the 2012 eg-ice Workshop.

Wülfing, A., Windisch, R., & Scherer, R. J. (2014). A visual BIM query language. eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2014, 157.

W3Schools (2020a). XML Attributes. https://www.w3schools.com/xml/xml_attributes.asp, accessed on 2020-11-05.

W3Schools (2020b). HTML Introduction. https://www.w3schools.com/html/html_intro.asp, accessed on 2020-11-05.

Xbim Toolkit (2020a). https://docs.xbim.net/, accessed on 2020-11-02.

Xbim Toolkit (2020b). Basic model operations. https://docs.xbim.net/examples/basic-model-operations.html, accessed on 2020-11-02.

Zhang, C., Beetz, J., & Weise, M. (2015). Interoperable validation for IFC building models using open standards. Journal of Information Technology in Construction (ITcon), 20(2), 24-39.