Technische Universität München

Ingenieurfakultät Bau Geo Umwelt

Lehrstuhl für Computergestützte Modellierung und Simulation

Extended Imperative Model Checking – A visual programming approach for a user-friendly MVD generation and validation

Bachelorthesis

für den Bachelor of Science Studiengang Umweltingenieurwesen

Autor:	Felix Sirtl
Matrikelnummer:	[redacted]
1. Betreuer:	Prof. Dr.-Ing. André Borrmann
2. Betreuer:	M.Sc. Jimmy Abualdenien
3. Betreuer:	Dr.-Ing. Simon Daum
Ausgabedatum:	15. April 2020
Abgabedatum:	29. September 2020

Acknowledgement

The result of this work in its entirety was significantly influenced by a number of people. I would like to name the most important of these below.

First of all, my supervisor from thinkproject Dr. Simon Daum should be mentioned. Very long discussions about concepts and implementations of the created application have strongly shaped the work. He also gave me an introduction to the world of programming. It has been an extreme enrichment for me to have the opportunity to work with him. It has always been a great pleasure for me.

My supervisor at the Technical University of Munich Jimmy Abualdenien has made a great contribution to my work with his explicit advice and suggestions. His ability to understand the context has been a major factor in shaping this work.

Furthermore, we thank Ferdinand Tausendpfund GmbH for providing their office building as a sample project.

Finally, I would like to thank my family and friends. Especially Lea Kaubisch and Simon Sirtl, who read the work and were able to give uninfluenced suggestions for improvement.

Abstract

Digital building models play a key role in the digitalisation of the building industry. In the Building Information Modeling (BIM) methodology, geometric and semantic information of the entire life cycle are mapped to a digital building model. This wealth of information makes automatically controlling the quality of the building information possible and essential. Industry Foundation Classes (IFC), the worldwide standard in the field of digital building with OpenBIM, provides the Model View Definition (MVD) method for implementing this automated quality assurance through the specification of the exchange requirements. The progressive extension of the MVD method and the technical implementation by means of the digital format mvdXML led to the fact that domain experts have difficulties in understanding and possible usage of MVDs.

This thesis attempts to give an overview of the complex MVD method, its implementation in the mvdXML format of buildingSMART and the integration in applications of different software vendors. Based on the knowledge of the analysis of the current method the aim of this work is created. The objective is to enable domain experts to create an MVD on their own through a user-friendly approach. Therefore, a concept that makes use of the widely established visual programming is presented. The visual programming approach is based on a general concept called Extended Imperative Model Checking (EIMC). The concept allows to analyse, filter, and validate data of an IFC model. The utilisation of visual programming is provided in the application EIMC-VP. It employs the imperative query language QL4BIM to interact with a building model. The application is intended to make it easier for the user to create and execute a query, respectively an MVD, in EIMC.

Zusammenfassung

Digitale Gebäudemodelle nehmen eine Schlüsselfunktion in der Digitalisierung der Baubranche ein. Innerhalb der Building Information Modeling (BIM) Methode werden in einem digitalen Gebäudemodell geometrische und semantische Informationen über den gesamten Lebenszyklus abgebildet. Die digitale Abbildung der Daten ermöglicht eine automatisierte Qualitätskontrolle dieser Informationen. Die Fülle an Informationen führt jedoch auch dazu, dass die Automatisierung für eine valide Kontrolle unerlässlich geworden ist. Der weltweite Standard des digitalen Bauens im OpenBIM Bereich, Industry Fundation Classes (IFC) stellt die Model View Definition (MVD) Methode zur Umsetzung der automatisierten Kontrolle mittels Austauschanforderungen zur Verfügung. Die fortschreitende Erweiterung der MVD Methode und deren technische Umsetzung durch das mvdXML Format führten dazu, dass ein domänenspezifischer Fachanwender Schwierigkeiten beim Verständnis und bei einer etwaigen Nutzung bekommt.

Diese Arbeit versucht zunächst einen Überblick über die vielschichtige MVD Methode zu gewähren. Dabei wird sowohl die Umsetzungen im mvdXML Format von buildingSMART als auch die Integration in Applikationen verschiedener Software Hersteller betrachtet. Das Ziel dieser Arbeit ist es daraufhin domänenspezifischen Fachanwendern zu ermöglichen selbstständig ein MVD nutzerfreundlich zu erstellen. Dafür wird ein Konzept vorgestellt, das sich der intuitiven Nutzung einer visuellen Programmiersprache bedient. Dieser Ansatz basiert auf dem erstellten allgemeinen Sprachkonzept Extended Imperative Model Checking (EIMC). Das allgemeine Konzept stellt Analysewerkzeuge zur Filterung und Validierung von IFC Modellen zur Verfügung. Die Anwendung des generellen Konzeptes erfolgt in dem entwickelten Prototyp der Applikation EIMC-VP. Die Applikation nutzt die imperative Abfragesprache QL4BIM, um mit einem IFC Instanz Modell zu interagieren. EIMC-VP soll dem Nutzer die Erstellung und die Ausführung einer Abfrage, respektive eines MVDs, in EIMC erleichtern.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AEC	Architecture, Engineering and Construction
BCF	BIM Collaboration Format
BIM	Building Information Modelling
BRep	Boundary Representation
EIMC	Extended Imperative Model Checking
GUID	Globally Unique Identifier
IDM	Information Delivery Manual
IFC	Industry Foundation Classes
JSON	JavaScript Object Notation
MVD	Model View Definition
QL4BIM	Query Language for 4D Building Information Models
STEP	Standard for the Exchange of Product model data
VP	Visual Programming
XML	Extensible Markup Language

1 Introduction

1.1 Motivation and objectives of the work

Building Information Model (BIM) has become an omnipresent term in the building industry. BIM aims to include every aspect of the building industry. Planning, execution, and management of buildings are associated with this term. Digital building twins, semantic information and processes between various actors involved in construction and facility management are reflected in BIM (Borrmann et al., 2015, p. 4).

With Industry Foundation Classes (IFC) the international non-profit organization buildingSMART has created a worldwide standard that aims to represent a realisation of the BIM methodology. The standard is extremely far-reaching and being continuously expanded (Borrmann et al., 2015, pp. 83 – 85). However, this all-encompassing schema is too far-reaching for specific issues. For this reason, Model View Definition (MVD) has been developed to delimit sub-areas which are sufficient for specific applications within the all-encompassing schema (buildingSMART, 2020e). Definitions of MVDs and the various application possibilities that result from it have grown with the expanding IFC schema. As a result, non-MVD experts face great obstacles to navigate through the cumulative terminology and definitions of the MVD topic. In recent years, a focus on one area of MVD use has crystallized: specializing in specific use cases (buildingSMART, 2019c). Besides far-reaching MVDs that cover almost the whole schema and are statically published by buildingSMART, the more specific MVDs are application-based focusing on comparatively small processes. These processes can often vary and must be adaptable. However, access to the use case specific MVDs is difficult, especially for actors who are not completely familiar with the topic (buildingSMART, 2019c).

The aim of this work is to provide an easier access to MVDs in general and the use case specific scenarios in particular. The creation of an MVD is considered essential for a better usability. In the course of this work a prototype of an application has been created which enables end users to create simple MVD requests themselves. The focus is on use case specific application cases, in particular the use case specific automated validation of information in a building model.

1.2 Structure of work

In order to approach the objective of this work - the creation of a new application and the underlying data model - in the best way, basic methods of the IFC schema are presented in Chapter 2. The selection is limited to areas relevant to the developed data model.

Chapter 3 explains the term Model View Definition and clarifies how it fits into the overall IFC schema. BuildingSMART's official technical implementation using the mvdXML file format is elucidated, and current scientific papers and software applications dealing with the topic are presented and evaluated. At the end of the Chapter a synopsis of previous statements and conclusions is given. The aim of the work is described in more detail.

In the following Chapter 4 Query Language for 4D Building Information Models (QL4BIM) is introduced. It forms the basis for the data model and the execution of the developed application.

In Chapter 5 the newly developed data model Extended Imperative Model Checking is presented. It is an extension of the QL4BIM data model and focuses explicitly on the MVD specific application.

Chapter 6 then illustrates the visual programming language EIMC-VP. This represents a way to use visual puzzle pieces to generate a query of the textual code and execute it on an instance model. It is intended to facilitate the usage for the end user.

In Chapter 7 the implementations are displayed. These are the implementations in QL4BIM and implementations of the application. Limitations of the concept described in Chapter 6 which are existing at the current state of the application are described here.

In the last Chapter 8 a conclusion and an outlook are given.

2 BIM – Building Information Modeling

BIM is a digital method of processing, storing, and exchanging information about the entire life cycle of a building (Borrmann et al., 2015, p. 4).

There are two fundamentally different approaches to BIM: Open- and ClosedBIM. ClosedBIM concentrates on one software and one model which performs all the work involved. OpenBIM, on the other hand, allows vendor-independent software in the editing process and focuses on a standard of exchange formats which should be usable by the various programs. An elementary problem of the ClosedBIM variant is that domain specific tasks often cannot be mapped explicitly enough (Borrmann et al., 2015, pp. 441). The use of uniform software products by all players in a construction project limits the diversity of the individual domains and specific workflows of the entire sector. In the OpenBIM approach, on the other hand, the free choice of software products aims to create the greatest possible freedom (Borrmann et al., 2015, pp. 440). At fixed points in time, different partial models of the individual domains are combined to form one coordination model. In order to be able to combine different models, a fundamental basis for the implementation of the individual models is required. The IFC format acts as such a basis and is the common standard for working in the OpenBIM area today. The wide range of applications of OpenBIM leads to an enormous amount of information that must be able to be represented in a data model that attempts to satisfy this. Technical implementations enable this wealth of information to be checked automatically. This ensures that the information stored in the model meets certain quality assurances. And within these different automated quality assurances, the MVD methodology, respectively the implementation through the mvdXML format, was developed by buildingSMART. As an introduction to the topic, this Chapter presents general basics of IFC in order to be able to build on them in the following Chapters.

2.1 Industry Foundation Classes

The IFC standard is unique in its current scope. It is a non-commercial digital data model for building models and has become the standard in the Architecture, Engineering and Construction (AEC) industry worldwide (Borrmann et al., 2015, p. 85; buildingSMART, 2020a). The non-profit organisation buildingSMART is the founder of the IFC format. The organisation aims to advance the open source standard. Objective of the developed files is to offer a concept that is as comprehensive as possible (buildingSMART, 2019a). BuildingSMART has decided to describe the schema with the data modelling language EXPRESS and the graphical notation of it EXPRESS-G. In the IFC data model, individual objects are called entities. It contains information about inheritance and relationships between individual entities, geometry, and the structure of alphanumeric attributes of the model. In addition, rules are set up to ensure that attributes can only be assigned to certain values. It provides all possibilities that can be realised by instance files in practicable application. The actual instance model can be stored in a schema-compliant file, which contains explicit values for a specific model. The instance file is usually encoded in a Standard for the Exchange of Product model data (STEP) physical file or a STEP-XML (Borrmann et al., 2015, pp. 84 – 85). The following sections present various concepts within the IFC schema relevant to the development of the approach.

2.1.1 Inheritance

In object-oriented modelling languages, various attributes between object classes are inherited. The IFC schema is - with its attempt to cover as many areas of construction industry as possible - a very complex concept (Borrmann et al., 2015, p. 88). For a better overview and expandability, different layers have been developed. The purpose of these layers is to show which classes can inherit attributes to which classes. If a layer is above a subordinate layer, it inherits attributes to the underlying layer, vice versa this possibility does not exist (Borrmann et al., 2015, p. 88). For example, each entity of IfcRoot has a GlobalID for unique identification, which IfcRoot inherits up to the entities of IfcWallStandardCase. The resource layer is different from the other three layers, none of the classes in this layer is derived from IfcRoot (Borrmann et al., 2015, p. 90). It follows that an object of this layer has no GlobalID - no identifier. Due to the absence of this attribute, a class from the resource layer cannot exist alone in the Ifc schema. The entities of this layer always need a link to an object that is derived from IfcRoot.

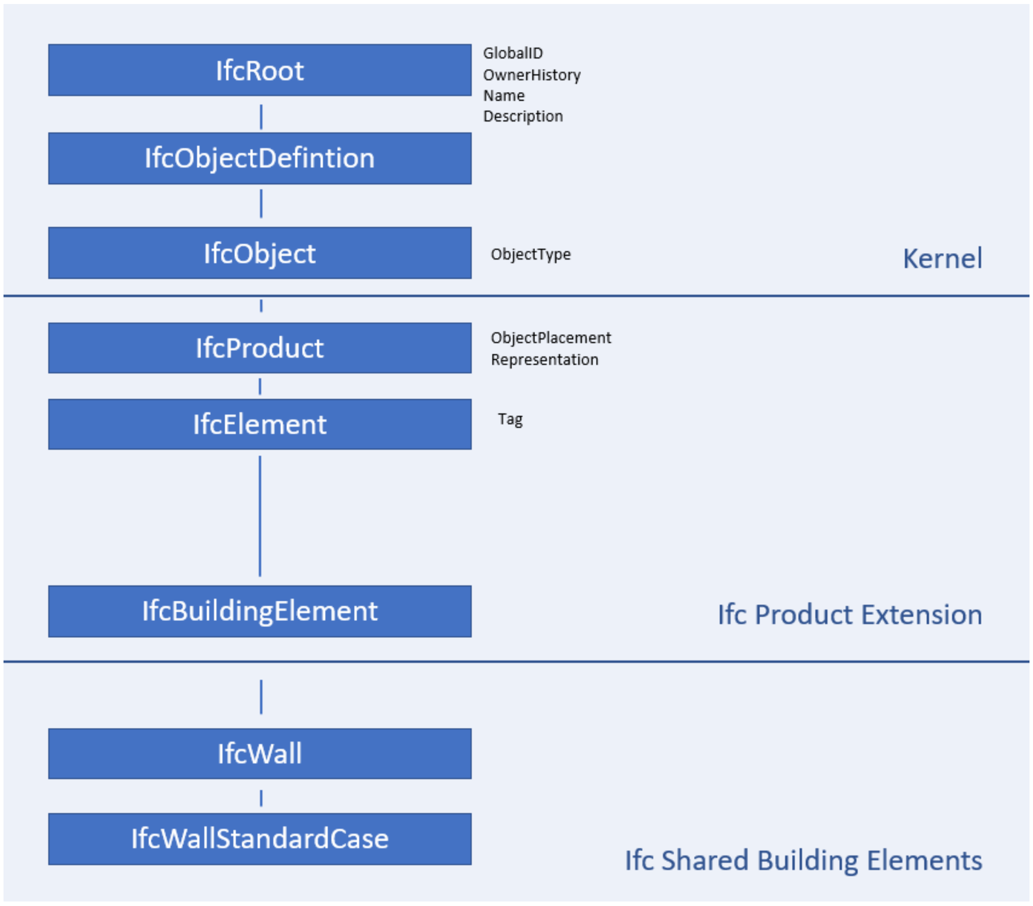


Figure 2-1 Layers and therein contained entities in IFC (Adopted from BIM Supporters B.V., 2018, 05:50)

2.1.2 Relationships

Entity relationships play an important role in the IFC data model. The relationships that are used to describe connections between entities in the IFC schema even stand out from other object-oriented modelling languages due to their level of detail (Borrmann et al. 2015, p. 93). The IFC schema follows objectified relationships. Objectified relationships do not represent connections as direct references, but as independent objects. So, relationships in the IFC schema represent object classes of their own. These connections are always aligned in a certain direction. In order to offer a possibility to ‘run’ the direction of the connection vice versa, conceptual inverse relations were introduced. These constitute the exact opposite direction. There are several different relationship entities. Analogous to an object, sub entities inherit attributes from a supertype class. Thus, the objectified relationships are always sub classes of the class IfcRelationship (Borrmann et al., 2015, p. 93).

2.1.3 Property Sets

Attributes are firmly anchored properties of an object class. To prevent an entity from being unnecessarily overloaded by unused attributes, property sets have been introduced (Borrmann et al., 2015, p. 112). A property set represents a container that holds semantically related properties. The possibility to create user-defined property sets also allows to individualise the data model without having to adapt the complete schema. Within a property set individual properties are assigned to the property set. BuildingSMART provides predefined property sets for a variety of uses (buildingSMART, 2020b). A typical predefined Property Set in IFC is Pset_WallCommon. It contains general information about entities of the class IfcWall and their subclasses. The property set offers, among other things, space to store information about acoustic rating, fire rating and indication whether it is an exterior or interior wall (buildingSMART, 2020c).

2.1.4 Geometry

In BIM both the semantics and the geometric representation of objects have a high priority. Both areas may need to be varied separately in terms of content and representation at different times in the life cycle of a building and in different domains. In order to take this into account, the IFC schema treats semantics and geometry separately. This strict separation is implemented in the schema by means of a separate class for geometries (Borrmann et al., 2015, pp. 100). Geometric representation entities are located in the resource layer. This means, they have no Globally Unique Identifier (GUID) and therefore cannot exist without an entity that inherits a GUID. GUIDs are unambiguously identifiers in the IFC schema. An IfcProduct can have none, one, or more geometries attached. This is especially important since different actors in the construction industry require different level of detail of a building. Different geometrical representations are supported in IFC. Among others, this support includes Constructive solid geometry, Boundary representation (BRep) and Boolean operations (Borrmann et al., 2015, pp. 101 – 102).

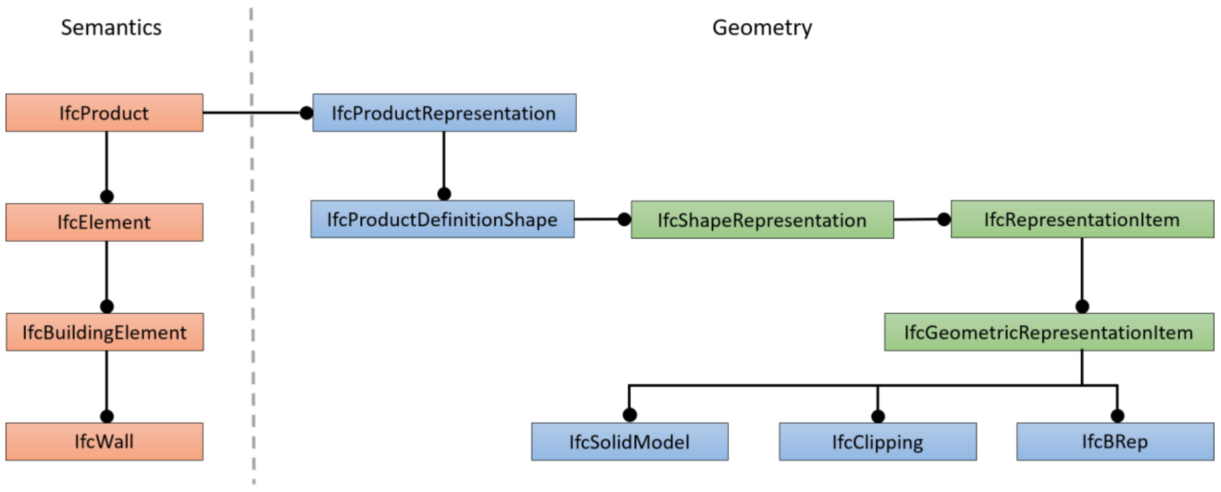
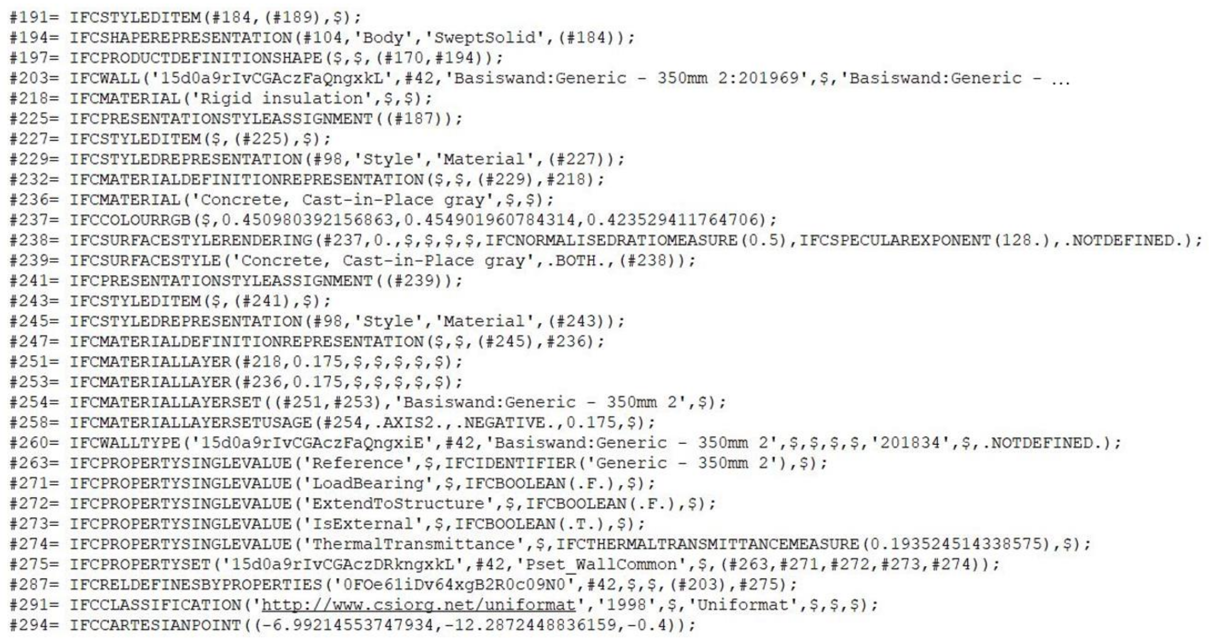


Figure 2-2 Visualization of the relation between semantical and geometrical representations in IFC (Adopted from Borrmann et al., 2015, p.101)

Figure 2-2 graphically illustrates how the semantic modelling of the aforementioned entity IfcWall can be linked to different geometric representations, like IfcSolidModel, IfcClipping or IfcBRep.

2.2 IFC instance models

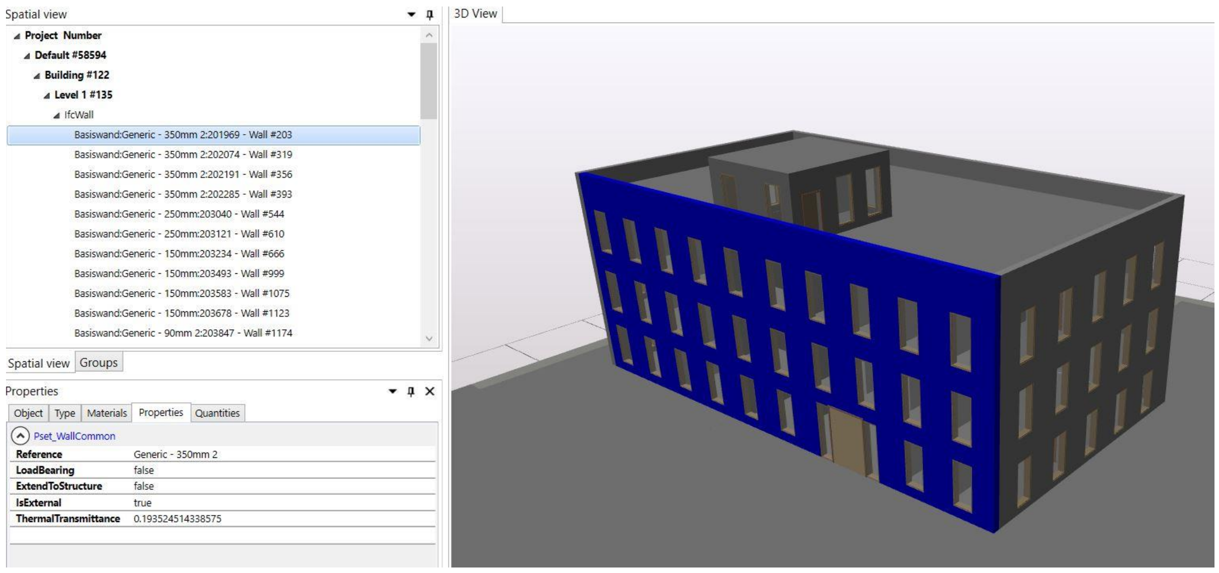
The IFC schema describes the structure and relationships of information. An instance file stores concrete building model data according to specifications given in the schema. This data is saved in a text file. The structure of the text file can follow different formats. The two most common formats are STEP and XML (Borrmann et al., 2015, pp. 87). The files formatted according to part 21 in the STEP specification are marked with the extension '.ifc'. Due to their compactness, they are the most widely used variant. Files which are formatted according to the XML schema are marked with the extension '.ifcXML'. They are particularly suitable when the file is used with XML tools. However, they have a larger file size compared to the STEP file. Even if the size of the file is different, the choice of the file format does not affect the information range stored in it. Both formats allow the information to be displayed to the same extent (buildingSMART, 2019b).



```
#191= IFCSTYLEDITEM(#184,(#189),$);
#194= IFCSHAPEREPRESENTATION(#104,'Body','SweptSolid',(#184));
#197= IFCPRODUCTDEFINITIONSHAPE($,$,(#170,#194));
#203= IFCWALL('15d0a9rIvCGAczFaQngxkL',#42,'Basiswand:Generic - 350mm 2:201969',$,'Basiswand:Generic - ...
#218= IFCMATERIAL('Rigid insulation',$,$);
#225= IFCPRESENTATIONSTYLEASSIGNMENT((#187));
#227= IFCSTYLEDITEM($,(#225),$);
#229= IFCSTYLEDREPRESENTATION(#98,'Style','Material',(#227));
#232= IFCMATERIALDEFINITIONREPRESENTATION($,$,(#229),#218);
#236= IFCMATERIAL('Concrete, Cast-in-Place gray',$,$);
#237= IFCCOLOURRGB($,0.450980392156863,0.454901960784314,0.423529411764706);
#238= IFCSURFACESTYLERENDERING(#237,0.,$,$,$,$,IFCNORMALISEDRATIOMEASURE(0.5),IFCSPECULAREXPONENT(128.),.NOTDEFINED.);
#239= IFCSURFACESTYLE('Concrete, Cast-in-Place gray',.BOTH.,(#238));
#241= IFCPRESENTATIONSTYLEASSIGNMENT((#239));
#243= IFCSTYLEDITEM($,(#241),$);
#245= IFCSTYLEDREPRESENTATION(#98,'Style','Material',(#243));
#247= IFCMATERIALDEFINITIONREPRESENTATION($,$,(#245),#236);
#251= IFCMATERIALLAYER(#218,0.175,$,$,$,$,$);
#253= IFCMATERIALLAYER(#236,0.175,$,$,$,$,$);
#254= IFCMATERIALLAYERSET((#251,#253),'Basiswand:Generic - 350mm 2',$);
#258= IFCMATERIALLAYERSETUSAGE(#254,.AXIS2.,.NEGATIVE.,0.175,$);
#260= IFCWALLTYPE('15d0a9rIvCGAczFaQngxiE',#42,'Basiswand:Generic - 350mm 2',$,$,$,$,'201834',$,.NOTDEFINED.);
#263= IFCPROPERTYSINGLEVALUE('Reference',$,IFCIDENTIFIER('Generic - 350mm 2'),$);
#271= IFCPROPERTYSINGLEVALUE('LoadBearing',$,IFCBOOLEAN(.F.),$);
#272= IFCPROPERTYSINGLEVALUE('ExtendToStructure',$,IFCBOOLEAN(.F.),$);
#273= IFCPROPERTYSINGLEVALUE('IsExternal',$,IFCBOOLEAN(.T.),$);
#274= IFCPROPERTYSINGLEVALUE('ThermalTransmittance',$,IFCTHERMALTRANSMITTANCEMEASURE(0.193524514338575),$);
#275= IFCPROPERTYSET('15d0a9rIvCGAczDRkngxkL',#42,'Pset_WallCommon',$,(#263,#271,#272,#273,#274));
#287= IFCRELDEFINESBYPROPERTIES('0FOe61iDv64xgB2R0c09N0',#42,$,$,(#203),#275);
#291= IFCCLASSIFICATION('http://www.csiorg.net/uniformat','1998',$,'Uniformat',$,$,$);
#294= IFCCARTESIANPOINT((-6.99214553747934,-12.2872448836159,-0.4));
```

Listing 2-1 IFC file opened with a text editor

Listing 2-1 shows an excerpt of an '.ifc' file opened with a text editor software. The file is representing the information of the contained model in a textual manner. The excerpt contains information about the link between the IsExternal property to a wall. Line #203 introduces a particular wall element in the IFC model. Various attributes are described by name directly in the same line, such as the GUID at the beginning of the expression in brackets. Relationships are realised with line numbers. For example, in line #287 the connection between the entity IfcWall and the property set Pset_WallCommon is established through the objectified relationship IFCRELDEFINESBYPROPERTIES. It refers to the IfcWall element in line #203 as well as to the property set Pset_WallCommon in line #275. The simple data type SingleValue in line #273, which represents the property IsExternal and its value TRUE, is attached directly via the line number to the property set. Within the argument of the property set four more numbers can be seen. These refer to other properties in the same property set, for instance LoadBearing. When editing IFC instance files, care must be taken that the file cannot be worked through once from top to bottom to obtain all possible information. In line #203 there is no reference to the following objectified relation in line #287, due to the fact that there is no description of inverse relationships within STEP files. Only in line #287 the wall is mentioned and only here the property set is referenced. When implementing algorithms for validation of IFC instance models, this must be considered. The following image shows the corresponding IFC file opened with XBIM. The external wall #203 is selected.



- Project Number
  - Default #58594
    - Building #122
      - Level 1 #135
        - IfcWall
          - Basiswand:Generic - 350mm 2:201969 - Wall #203
          - Basiswand:Generic - 350mm 2:202074 - Wall #319
          - Basiswand:Generic - 350mm 2:202191 - Wall #356
          - Basiswand:Generic - 350mm 2:202285 - Wall #393
          - Basiswand:Generic - 250mm:203040 - Wall #544
          - Basiswand:Generic - 250mm:203121 - Wall #610
          - Basiswand:Generic - 150mm:203234 - Wall #666
          - Basiswand:Generic - 150mm:203493 - Wall #999
          - Basiswand:Generic - 150mm:203583 - Wall #1075
          - Basiswand:Generic - 150mm:203678 - Wall #1123
          - Basiswand:Generic - 90mm 2:203847 - Wall #1174

Pset_WallCommon

Reference	Generic - 350mm 2
LoadBearing	false
ExtendToStructure	false
IsExternal	true
ThermalTransmittance	0.193524514338575

Figure 2-3 Visual representation of the ifc-file opened with Xbim Xplorer

3 Process-based exchange requirements

In the previous Chapter 2 the IFC data model was introduced. It can provide all information related to a building model throughout its total life cycle. The possible integration of all information in only one model leads to the fact that this model can become very complex. However, it is of major importance for efficient work to receive only the information that is relevant to oneself and to know which information has to be passed on to which parties at what exact time. In addition, the life cycle is a process in which different parties may have only access to different information at different times. Therefore, buildingSMART has developed informational provisions and technical arrangements to meet these process-based challenges (Borrmann et al., 2015, pp. 130 – 133). The provision is realised in Information Delivery Manuals (IDM) and with Model View Definitions (MVD). The IDM methodology specifies processes and information flows within the life cycle of a building (buildingSMART, 2020d). The technical implementation of an IDM is then realised in a Model View Definition. IDM is briefly presented in the next section, followed by a more detailed look at how it is implemented in MVDs. The usage of MVDs through mvdXML is reviewed and the Chapter is concluded with a synopsis of MVDs.

3.1 IDM and MVD method

IDM is a methodology to describe work processes in the IFC data model in order to facilitate interoperability between different parties and improve efficiency of workflows (buildingSMART, 2020d). The development of IDM and MVD is carried out by domain and technical experts. At the beginning of the elaboration it is determined which basic improvements and process aspects are assessed. Then an overview of the sub-process which will be specified is created. Participants, their roles, and tasks are defined. Also, the extraction of repetitive processes and information to be exchanged are determined. Once this has been declared, the individual Exchange Requirements are listed in a table form (Borrmann et al., 2015, pp. 131 – 132).

MVDs technically apply information gathered in the IDM. Therefore, it is necessary to further formalise the ERs within the IDM to allow a mapping to the IFC schema (Borrmann et al., 2015, p. 134). An MVD usually combines several ERs that belong to one project. Moreover, MVDs are also used today without a previous IDM specification.

They can be exploited for a variety of tasks. For instance, an MVD enables an automatic data quality assurance in an IFC model (Chipman et al., 2016). The applications and purposes of MVDs are discussed in more detail in the following sections 3.2 and 3.4.

BuildingSMART publishes official MVDs. Software manufacturers can have their products certified by buildingSMART on one or more official MVDs. This ensures that the certified software products can correctly handle the classes and entities contained in the IFC sub schema described by a specific Model View. An important part of this certification is to ensure that the products can correctly import and export IFC-files, in range of the MVD (Borrmann et al., 2015, pp. 136 – 137). There are four official MVDs for the IFC2x3 version. In the latest schema version IFC4 ADD2 only two official MVDs are available at the moment, but four more are already announced and in progress (buildingSMART, 2020f). The two existing ones are the Reference View and the Design Transfer View. First one is intended to coordinate model information between the architectural, structural, and building services domains. A simplification of geometrical and relational representations takes place, as no changes in geometry of the model after export are intended. The Design Transfer View represents a transfer of further geometric and relational information of a digital model. In case of further editing the model, the Design Transfer View represents a transfer of extended geometrical and relational information. However, not all possible entities and relations of a model are transferred. Specified processing possibility is ensured, but a complete, all-encompassing transfer of a model is not carried out (buildingSMART, 2020e).

3.2 MVD purpose

The IFC schema aims to cover all the different domains and areas in the AEC industry. Cross-domain and generally valid solutions, as described in the IFC schema, can lead to confusion and unmanageability in process workflows. Therefore, the conceptual purpose of a MVD is to limit the scope of the entire IFC schema to a particular subset that is considered sufficient for certain exchange scenarios (buildingSMART, 2020e). BuildingSMART delivers with mvdXML a possibility for converting the MVD concept into an electronic format. This format makes it easier to understand applications and associated implications of the abstract MVD purpose. The objectives of mvdXML are mainly (Chipman et al., 2016):

- to tailor the total IFC schema to subset schemas

- to document the implementation and usage of such subsets
- to support filtering of IFC instance files
- to sustain quality assurance through validation of IFC data sets

The idea to reduce the IFC schema to subsets for exchange explicit requirements is determined in the first vignette. The following points are related to it. In the documentation, the use of a subset is described in detail. The essential elements of the defined subset can be shown in instance diagrams. The documentation of the implementation and usage of a subset is especially important for software developers for interpretation of the IFC specification (Chipman et al., 2016). The filtering of instance models results in a model subset. Analogous to the schema reduction, the scope of information is limited. Validation enables the data to be checked. For example, constraints can be checked. Certain restrictive values that are valid for attributes and properties, or certain geometric representation variants (Borrmann et al., 2015, p. 136).

3.3 mvdXML

In this section a more detailed representation of mvdXML is given. After surveying the underlying XML format and mvdXML schema, special emphasis is put on the two essential tasks filtering and validation. Thereupon an assortment of current works illustrates the popularity of mvdXML. The selection also demonstrates the importance of filtering and validation using mvdXML.

3.3.1 XML

Extended Markup Language (XML) is a mark-up language that has achieved wide acceptance (Borrmann et al., 2015, p. 124). Since its invention, the XML format has gained significant importance in data exchange via the Internet (World Wide Web Consortium, 2016). XML offers the possibility to design a mark-up language completely independently from any pre-settings. In this language, data can be represented in a structurally logical way. For this purpose, *elements* that provide the semantic division of the content are used. Start and end of an *element* is defined by tags. There are three types of tags:

- start tag: `<tag>`
- end tag: `</tag>`
- empty tag: `<tag/>`

The content of the elements is described within the tags and it can contain text, attributes, and *elements*. This enables the construction of logical structures by nesting different elements into each other to describe the data to be exchanged. Hence, XML does not represent commands as a programming language would. Instead, it is a descriptive representation of the content, which makes it descriptive programming. A file written in XML language is considered a universal data format (Saake et al., 2018). Thus, it offers the possibility of exchanging data between different users. The mvdXML format makes use of this possibility. With standardized elements and hierarchies, the mvdXML schema is based on XML schema.

3.3.2 mvdXML schema

The basic structure of an mvdXML consists of two separate parts. Both parts and thus all others sub-parts in the schema start from the single valid root element mvdXML. The first part called Templates contains reusable ConceptTemplates and defines subgraphs of the IFC schema. The second part, the Views, describes how the previously defined parts and additional specification should be applied in MVDs to provide ERs (Chipman et al., 2016).

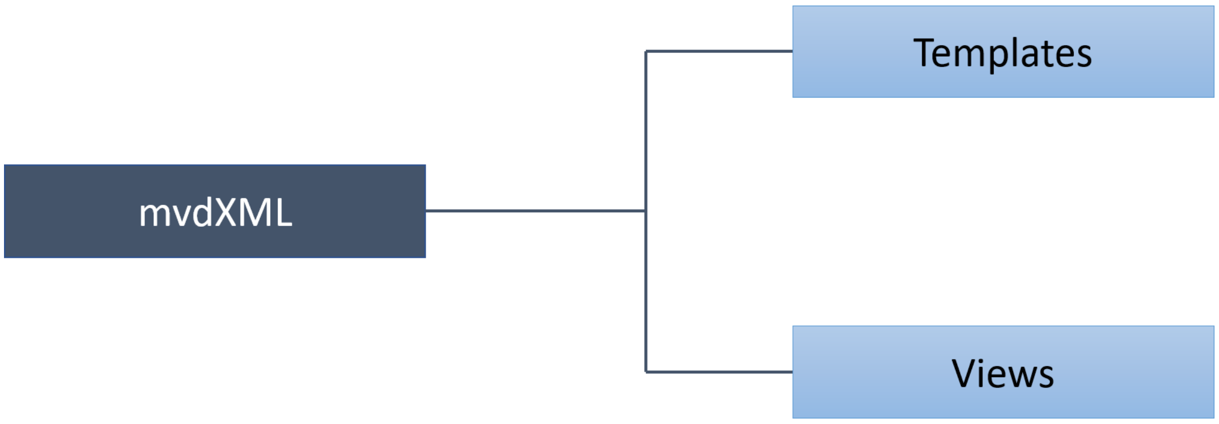


Figure 3-1 MvdXML two main parts Templates and Views

One or many ConceptTemplates can be defined in Templates. Each of these depicts a specific part of the IFC schema relevant to a particular functional unit. The starting point is an applicable entity, usually a subtype of IfcObject, followed by an alternate definition of AttributeRules and EntityRules. Depending on the focus of a template, these rules can be executed several times in succession. Constraints to certain schema areas, relationships or very explicit properties and their values can be specified with the rules. The result is a tree structure that provides the partial schema for a specific unit of functionality. Templates are prepared to allow easier handling of the constraints in the following Views part. The idea is to use templates across MVDs, so that recurring concepts can be modified to suit special use cases with less effort. Subsequently, the Views tag defines one or many MVDs. The ModelView determines specific ERs by using the previously defined ConceptTemplates. It is also possible to intense the scope of the predefined templates. ExchangeRequirements state, if and how elements defined in the ConceptRoot tag are applied in a specific exchange scenario. In that ConceptRoot, the tag Applicability offers space for further constraints, these can be referenced on ConceptTemplates. In Concepts the logical statements are then defined in TemplateRules using parameters in order to be able to carry out validations. It should be emphasized that the mvdXML file often provides many ways to represent identical information, due to the fact that it tries to cover a wide range of different use cases. Figure 3-2 is a simplified representation of the structure of an mvdXML (Chipman et al., 2016).

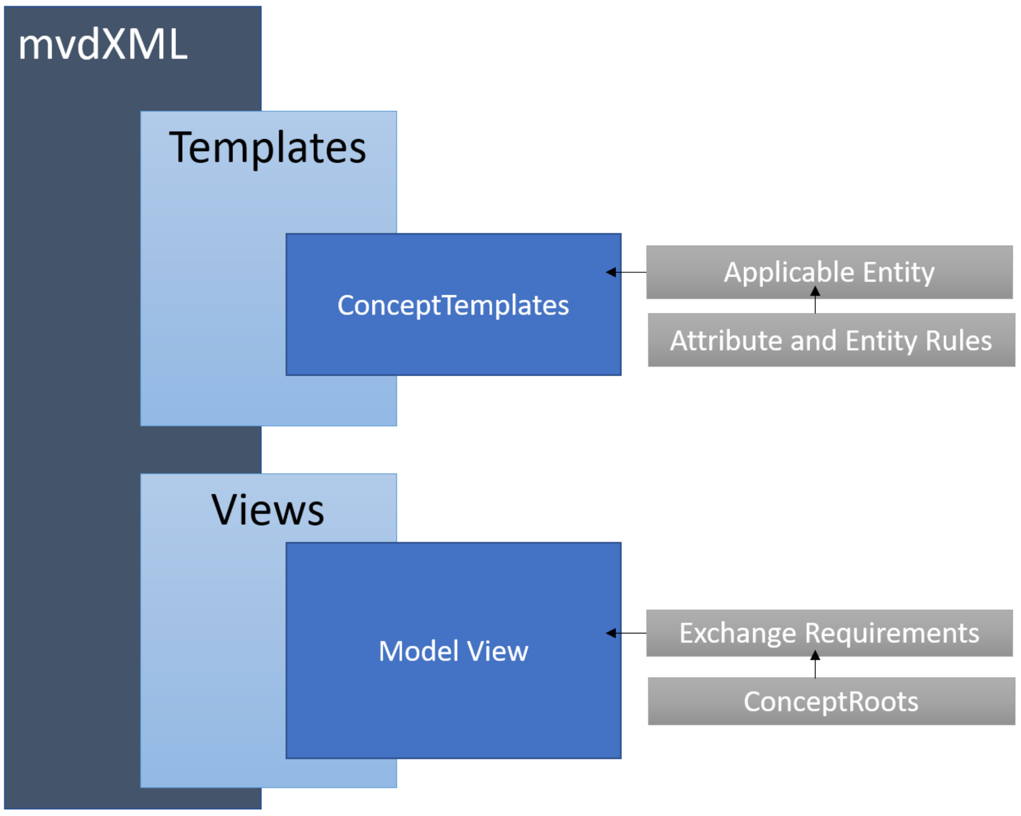


Figure 3-2 Simplified mvdXML schema (Adopted from Chipman et al., 2016)

3.3.3 Filter and Validation of IFC models with mvdXML

MvdXML uses the structure of XML to define required entities and map them to the Ifc schema in order to document ERs, specify subset schemas of the entire IFC schema and perform filter tasks and validations with parameter constraints. Within the ConceptTemplates described in section 3.3.2, structures are built that can be reused in different MVDs. They map the structure of the IFC model and enable access to it. In the Views tag previously defined templates are referenced and functions can be executed. On the one hand, the schema can be further reduced to the required specifications and on the other hand validations can be performed. The following example from the latest mvdXML 1.1 specification shows the structuring of a sub-graph of the IFC schema, represented in mvdXML. An example from the official guideline was deliberately chosen in order to be representative of the underlying structural features. The full mvdXML is attached in the appendix. The mapping is defined in two locally separated parts. In the Applicability of the Views tag it is possible to define a sub-graph. Since Concepts refer to ConceptTemplates within the Templates, sub-graphs are also defined by ConceptTemplates. All subgraphs combined form the subset schema for the specific MVD. It is important that the subset schema itself is a valid schema and has no inconsistencies (Chipman et al., 2016).

```
<!--- Header ---!>

 <Templates>
     <ConceptTemplate uuid="bafc93b7-d0e2-42d8-84cf-5da20ee1480a" name="Port Assignment"
      applicableSchema="IFC4" applicableEntity="IfcDistributionElement">
       <Rules>
            <AttributeRule AttributeName="IsNestedBy">
                <EntityRules>
                    <EntityRule EntityName="IfcRelNests">                  Section A
                        <AttributeRules>
                        <AttributeRule AttributeName="RelatedObjects">
                            <EntityRules>
                                <EntityRule EntityName="IfcDistributionPort">
                                    <AttributeRules>
                                        <AttributeRule AttributeName="Name" RuleID="Name"/>
  Section B                             <AttributeRule AttributeName="PredefinedType" RuleID="Type"/>
                                        <AttributeRule AttributeName="FlowDirection" RuleID="Flow"/>
                                    </AttributeRules>
                                </EntityRule>
                            </EntityRules>
                        </AttributeRule>
                        </AttributeRules>
                    </EntityRule>
                </EntityRules>
            </AttributeRule>
       </Rules>
     </ConceptTemplate>
 </Templates>
```

Listing 3-1 ConceptTemplate part of an mvdXML file representing recurring parts (Adopted from Chipman et al., 2016)

Section A in Listing 3-1 of the mvdXML file shows the connection how a distribution element is associated with a distribution port. Starting from the entity IfcDistributionElement, via the objectified relationship IfcRelNests to IfcDistributionPort.

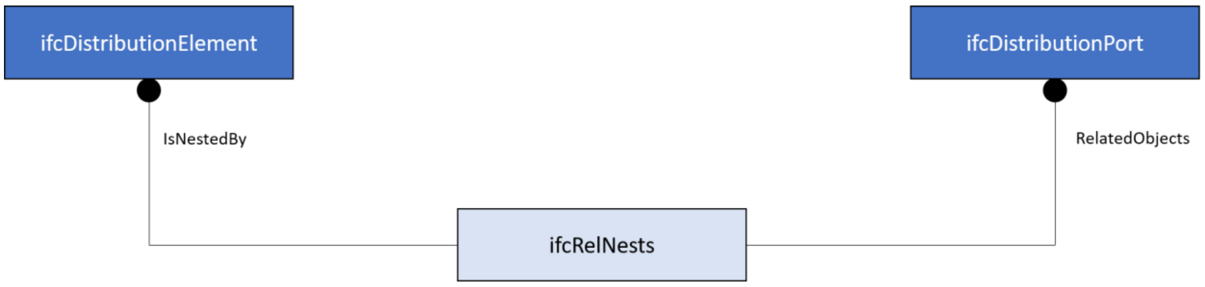
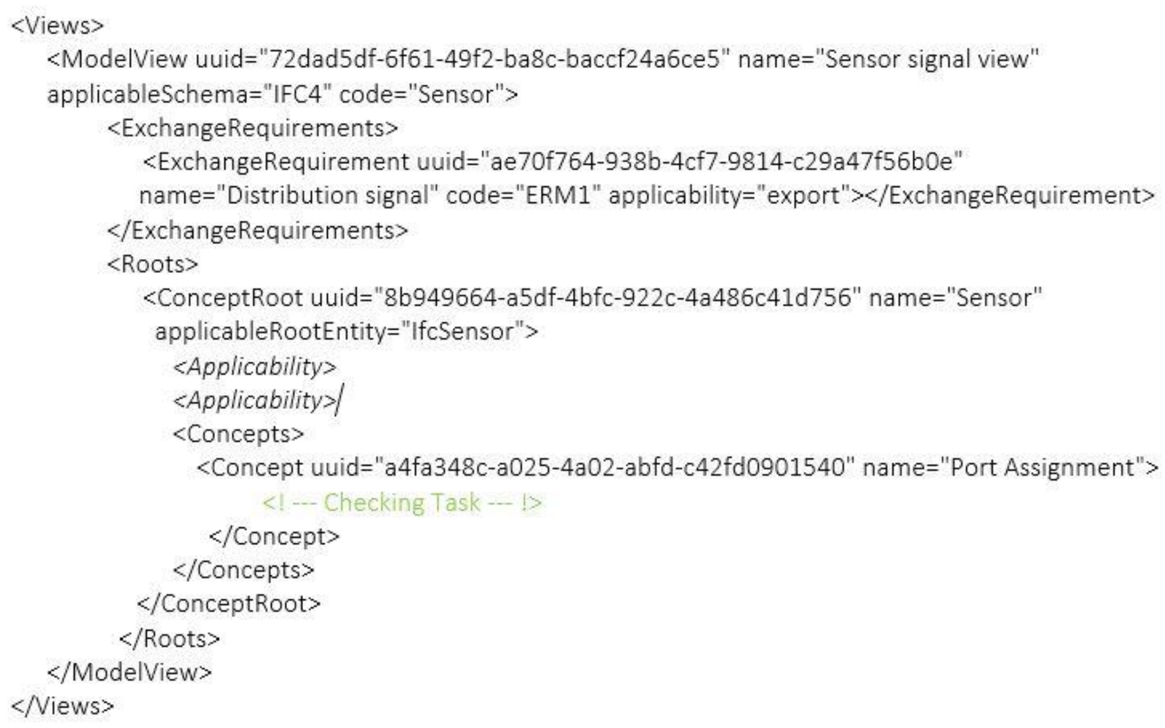


Figure 3-3 Representing Section A in an EXPRESS G Diagram

Further, in section B of Listing 3-1 the entity selection identified before is extended by providing certain attributes of the entity IfcDistributionPort with a RuleID. The RuleID allows specific access to the attributes in the Views.



```
<Views>
   <ModelView uuid="72dad5df-6f61-49f2-ba8c-baccf24a6ce5" name="Sensor signal view"
   applicableSchema="IFC4" code="Sensor">
        <ExchangeRequirements>
           <ExchangeRequirement uuid="ae70f764-938b-4cf7-9814-c29a47f56b0e"
           name="Distribution signal" code="ERM1" applicability="export"></ExchangeRequirement>
        </ExchangeRequirements>
        <Roots>
           <ConceptRoot uuid="8b949664-a5df-4bfc-922c-4a486c41d756" name="Sensor"
            applicableRootEntity="IfcSensor">
              <Applicability>
              <Applicability>
              <Concepts>
                <Concept uuid="a4fa348c-a025-4a02-abfd-c42fd0901540" name="Port Assignment">
                      <! --- Checking Task --- !>
                  </Concept>
              </Concepts>
           </ConceptRoot>
         </Roots>
   </ModelView>
</Views>
```

Listing 3-2 Views tag in a mvdXML file containing the Exchange Requirements (Adopted from Chipman et al., 2016)

In Views, the subgraphs defined in the ConceptTemplates are used to refine the filtering and to define the validation. A further filtering of the IFC schema can be done based on the ConceptTemplates in the Applicability tag. For this, the template is referenced and further restricted with TemplateRules. However, this further restriction does not have to follow, and the example above has no unit within the Applicability. Whereby the validation is applied to the entities named in applicableRootEntity and the connected template, the actual validation is performed within the TemplateRules. The basis for validation is given in the following XML excerpt. It is checked whether each sensor of the IFC model to be checked has at least one connector that sends signals.

```
<ConceptRoot uuid="8b949664-a5df-4bfc-922c-4a486c41d756" name="Sensor"
applicableRootEntity="IfcSensor">
<Concepts>
<Concept uuid="a4fa348c-a025-4a02-abfd-c42fd0901540" name="Port Assignment">
        <Template ref="bafc93b7-d0e2-42d8-84cf-5da20ee1480a"/>
        <Requirements>
                      Requirement exchangeRequirement="ae70f764-938b-4cf7-9814-c29a47f56b0e"
                      requirement="mandatory" applicability="export"/>
        </Requirements>
        <TemplateRules>
                    <TemplateRule Parameters="Name[Value]='Output' AND Type[Value]='SIGNAL' AND
                      Flow[Value]='SOURCE'" Description="Transmits signal."/>
        </TemplateRules>
</Concept>
<Concepts>
</ConceptRoot>
```

Listing 3-3 MvdXML *Concepts* part containing TemplateRules

One or more ConceptRoots are declared in the Roots. Each ConceptRoot refers an entity within the IFC schema. In the case above the entity IfcSensor is examined in the ConceptRoot tag. The reference to a ConceptTemplate expresses the representation within the IFC schema. The validation is now initialized by the expression mandatory in the ExchangeRequirements and described by TemplateRules. The parameters draw their information from the referenced template of the Concept. The RuleIDs defined in the ConceptTemplates (section B in Listing 3-1) are used to enable identification within the IFC schema. Various parameters can be linked with each other using logical operators (AND, OR, XOR). Via the relationship of IfcDistributionElements to IfcDistributionPorts established in section A in Listing 3-1, the entity IfcSensor - which is a subclass of IfcDistributionElements - inherits the attributes of IfcDistributionPorts. These are checked for exact name, type and value in the TemplateRules. If the naming of the attributes which is to be checked in an IFC model does not comply with the rules, the validation will display an error.

3.3.4 IfcDoc

The MVD just shown in the previous section could be created with a text editor but also with various applications. IfcDoc is a program that tries to facilitate the creation of an MVD. The main tasks are the creation of MVDs and the name-giving documentation of MVDs. The latest Version 12.2 of IfcDoc is used for this bachelor thesis. Furthermore, the IFC4 Addendum 2 Baseline for the IFC schema is used. IfcDoc offers a user interface that allows the user to create new MVDs based on the IFC schema. Extensive knowledge of inheritance, entities and their attributes in the schema is required to work with the program.

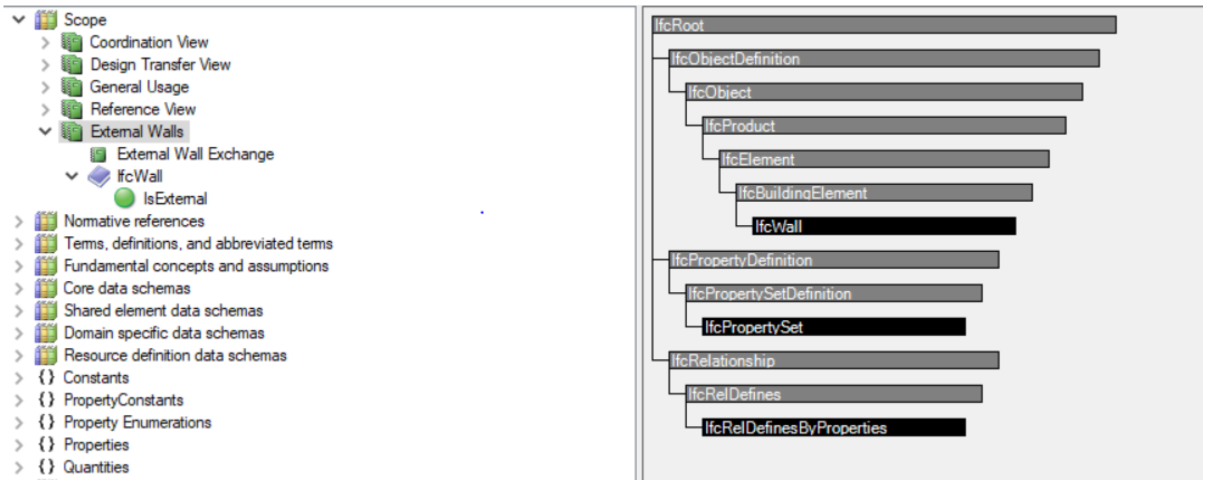
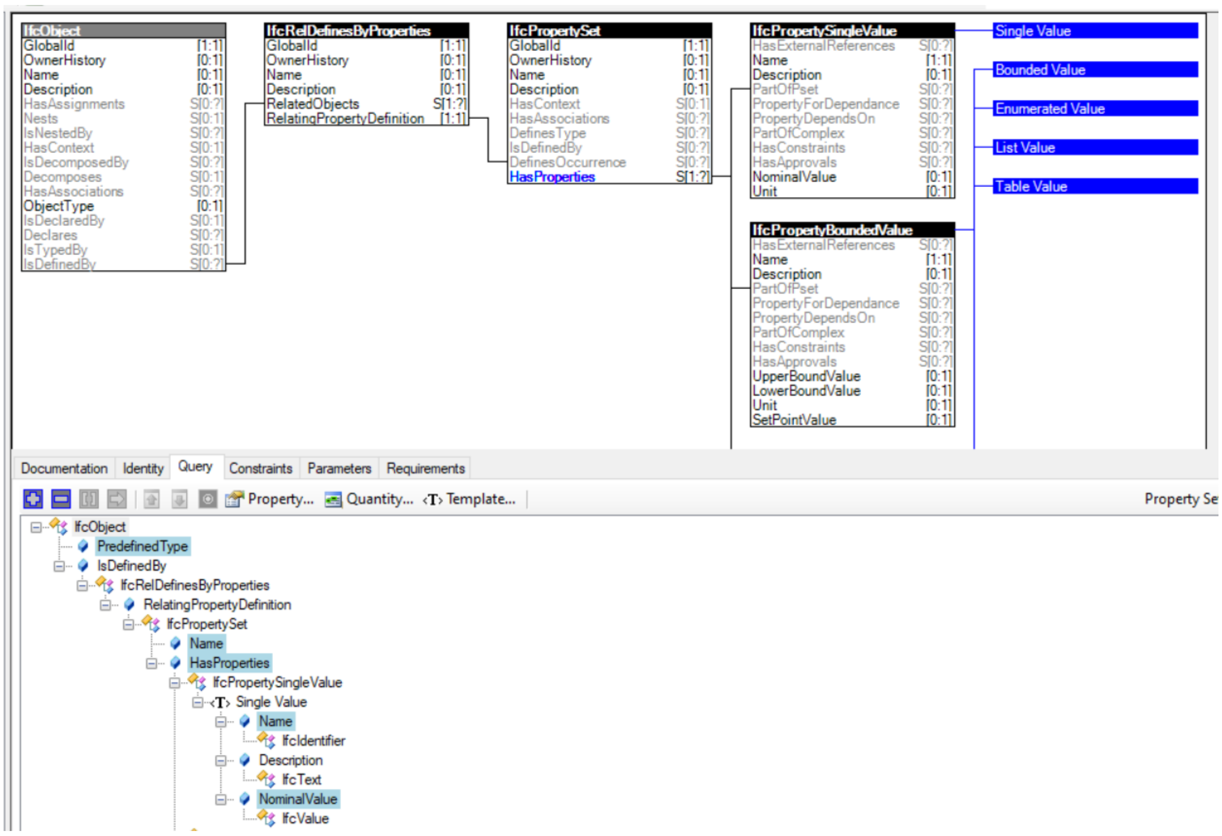


Figure 3-4 IfcDoc User Interface

Usage

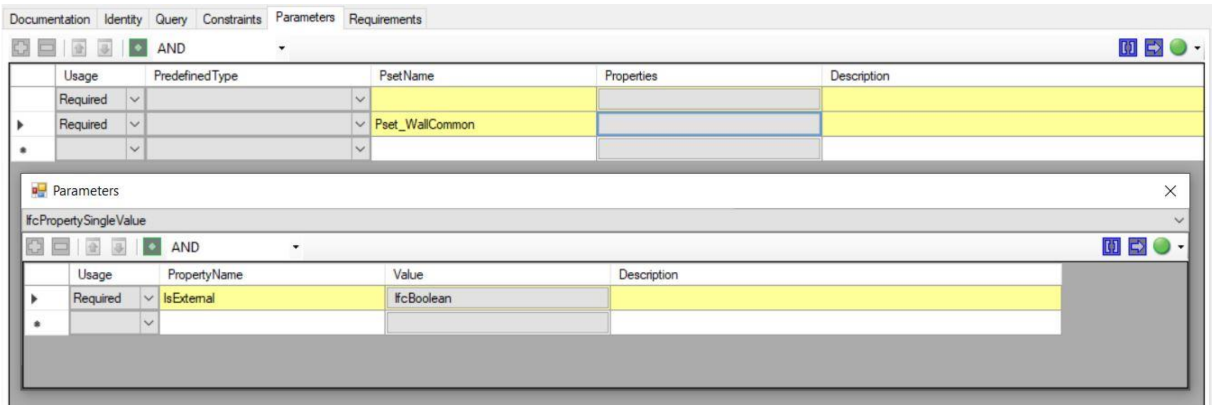
In IFC4 Addendum 2, there are already MVDs provided by BuildingSMART (Coordination View, Design Transfer View, General Usage, and Reference View). These can be used as a basis for incorporating extensions. It is also possible to create an MVD from scratch. For this work a simple MVD is to be constructed which contains only entities from IfcWall. Furthermore, the filtered walls shall be specified by selecting only walls with the property IsExternal and the value yes for this property. For this purpose, the entity IfcWall was included in the MVD and assigned to the exchange request 'External Wall Exchange'. In a so-called Column Definition certain requirements for IfcWall can be assigned. These requirements are already predefined in templates but can also be created based on the IFC schema itself.



```
IfcObject
  PredefinedType
  IsDefinedBy
    IfcRelDefinesByProperties
      RelatingPropertyDefinition
        IfcPropertySet
          Name
          HasProperties
            IfcPropertySingleValue
              Single Value
                Name
                  IfcIdentifier
                Description
                  IfcText
                NominalValue
                  IfcValue
```

Figure 3-5 IfcDoc templates

When selecting a template already stored in IfcDoc (Figure 3-5), the program independently creates the necessary relationships in the schema to add the new entity. In addition to a tabular list, the structure is displayed in EXPRESS-G format as a visual representation.



	Usage	PredefinedType	PsetName	Properties	Description
	Required				
▸	Required		Pset_WallCommon		
*					

	Usage	PropertyName	Value	Description
▸	Required	IsExternal	IfcBoolean	
*				

Figure 3-6 IfcDoc parameters template

By selecting the template, the application also creates the necessary parameters to be able to make restrictions (Figure 3-6). However, these restrictions could only be changed or extended minimally. For example, it was not possible to set the value of

IfcBoolean to yes or no. The created MVD consists - following the mvdXML format - of two parts. The first part contains the concept templates and the second part contains the model views.

Within the framework of this bachelor thesis restrictions could only be processed to a small degree within IfcDoc. The created MVD does not contain a rule that focuses on the selected property IsExternal. With knowledge of the structure and syntax of XML files and the ontology of the IFC schema, the MVD created from IfcDoc can be extended to meet the desired conditions in a text editor. The created MVD has been specified by editing an extension in the TemplateRule parameters. The name of the property on IsExternal and its existence with the attribute value yes. The original rule only provided the reference to the Pset_WallCommon.

Evaluation

The current version of IfcDoc has serious flaws and limitations that make it difficult to use. Within the scope of the work it was not possible to create rule sets with IfcDoc that allowed to design the filtering and validation capabilities of MVDs in a way that only considers certain entities with certain attributes and their attribute values. TemplateRules had to be subsequently incorporated with a text editor to enable these functionalities. So IfcDoc can be used to create a basic framework for a syntactically correct mvdXML. However, in most cases the user has to extend this framework manually in order to map important features. This manual extension requires an extended knowledge of the IFC schema itself and the mvdXML format in particular. This significantly restricts the use of IfcDoc for creating mvdXML files. IfcDoc requires a far-reaching revision to meet the needs of the user. It should also be noted that IfcDoc can be more efficiently used for documentation purposes. The generation of IFC instance diagrams is a great help for documentation. For this instance, it is more important that elements are represented in diagrams and that processes are described, rather than having to explicitly perform the individual processes (Chipman et al., 2016).

3.3.5 Current use cases of mvdXML in scientific research

MvdXML format is the official standard of BuildingSMART for formatting MVDs. Several researchers are working on different projects relating to mvdXML. In order to prove the popularity and various use cases of the mvdXML format, four examples that support the use of mvdXML are briefly described below.

Abualdenien and Borrmann (2019) developed a meta model design to improve the collaboration of different disciplines with regard to data consistency between different early stages of building phases. MvdXML is used to check the data consistency between these phases (Abualdenien, & Borrmann, 2019).

In another paper Abualdenien and colleagues (2019) investigate the automatic validation and subsequent filtering of IFC models for fire-safety and safe evacuation of pedestrians using mvdXML files. In a first step Bavarian laws were mapped to IFC. Secondly, in a pedestrian simulator the checked model is filtered for the information relevant for pedestrian evacuation using the same mvdXML file (Abualdenien et al., 2019, pp. 191 – 198).

Zhang and colleagues (2015) developed a Model View Checker. The prototype validates IFC instance models based on mvdXML rules and then generates a report in BIM Collaboration Format (BCF) to document the issues identified (Zhang et al., 2015).

Baumgärtel and colleagues (2016) developed a method to convert already existing mvdXML files into IFC Query Language (ifcQL). Using the generated ifcQL code a filtering and validation on instance level is possible (Baumgärtel et al., 2016).

3.3.6 mvdXML conclusion

MvdXML offers a wide range of possibilities due to the extensible and adaptable XML schema. These various possibilities and the open source usability lead to mvdXML being used in many current scientific projects (see section 3.3.5). Nevertheless, there are some problems associated with the diversity of mvdXML. The extracts of the MVD shown in this Chapter can be mapped in mvdXML in different ways. Thus Zhang and colleagues (2015) claim that "[…] mvdXML is a semi-structured description method rather than a logic-based strictly formatted method" (p. 26). In connection with the pure descriptive method of XML, it leads to the fact that names and not the semantics of Concepts are used for representing and referencing Concepts. A strict guidance and possibly further narrowing of the mvdXML schema could provide an easier introduction and lead to greater reusability (Zhang et al., 2015).

3.4 Commercial MVD Tools

Besides IfcDoc, there are already programs from various commercial software vendors offering similar functionalities. SimpleBIM which will be examined in detail below, BIMQ, and properBIM are a selection of such commercial tools. Afterwards, a general evaluation of the examined programs is presented.

3.4.1 SimpleBIM

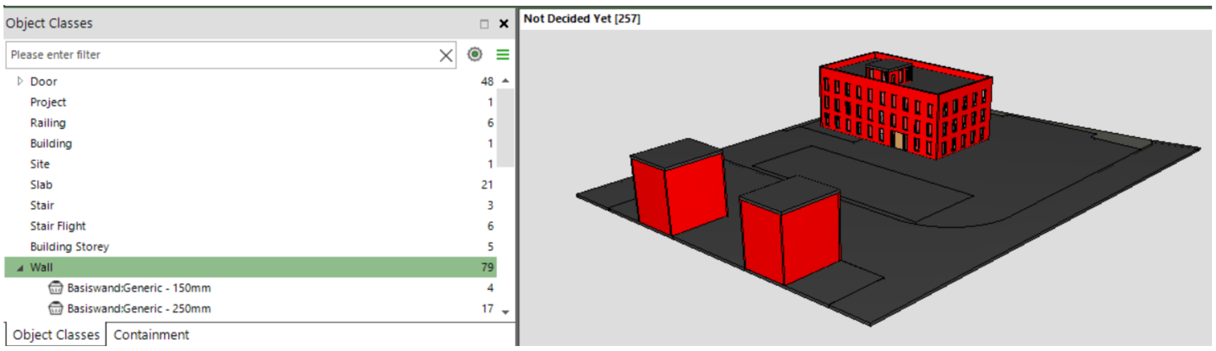
With SimpleBIM, the company Datacubist provides an application that reflects features of IfcDoc and adds new aspects. Various usages can be carried out with the software SimpleBIM. The user can edit and validate an IFC model and exclude relevant areas. Validation and reduction to a specific subset are essential features of an MVD. Compared to IfcDoc the advantage of SimpleBIM is its user-friendliness. Intuitive methods such as drag and drop allow the user to work directly on the model. This permits a much faster entry into the work with ERs than IfcDoc can offer it. Furthermore, no expert knowledge of the IFC schema or XML is required. In addition, you can add information to the IFC instance model. For instance, properties can be added. In order to shorten the user's workflow, all processes can be automated with templates (SimpleBIM, 2020a). In this work version 8.2 with a free trial access was used to examine the application.

Templates

SimpleBIM offers templates, which can be used to define ERs, so that the user does not have to formulate requirements from scratch. Furthermore, the editing process can be automated. Templates are defined excel tables. A predefined pattern needs to be followed so that they can be used correctly. In order to follow this pattern, SimpleBIM (2020b) recommends that you make your own settings based on predefined Excel tables. However, a template can also be created independently and from beginning, this requires a certain amount of familiarisation and experience. To be able to use more advanced techniques, groups can be created within a template. Groups form a simple tool for organizing objects or persons that do not exist as a grouping in the IFC schema. Likewise, several templates can be connected in series to create a sequence of several automated steps.

<u>Reduction to the relevant</u>

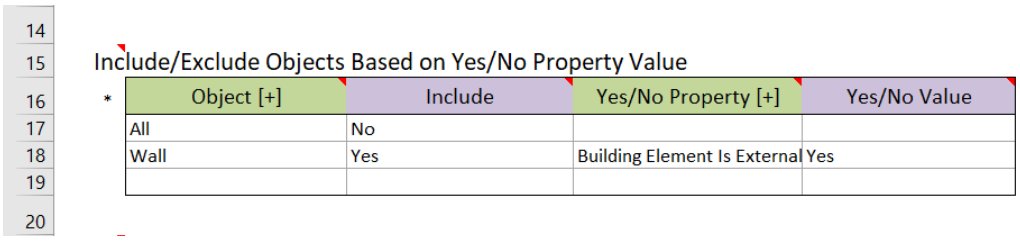
One of the main tasks of SimpleBIM is to eliminate irrelevant information - for a given task - and thus to focus the model explicitly on a specific workflow. The Model Trimmer allows you to restrict a building to relevant objects without a deeper basic knowledge of the IFC schema. Objects can be selected and highlighted using a folder structure.



Door	48
Project	1
Railing	6
Building	1
Site	1
Slab	21
Stair	3
Stair Flight	6
Building Storey	5
Wall	79
Basiswand:Generic - 150mm	4
Basiswand:Generic - 250mm	17

Figure 3-7 SimpleBIM selecting all walls

Property sets can also be accessed and thus the selection of relevant objects can be specified. For example in Figure 3-7, all walls with the property IsExternal and the value yes can be manually selected. Within SimpleBIM, the IsExternal property has been renamed by Datacubist to Building Element is External. This renaming is intended to make it easier for the user to understand the purpose of the property (IsExternal represents a property within Pset_WallCommon and assigns a component whether it has a point of contact with the exterior space.).


Include/Exclude Objects Based on Yes/No Property Value

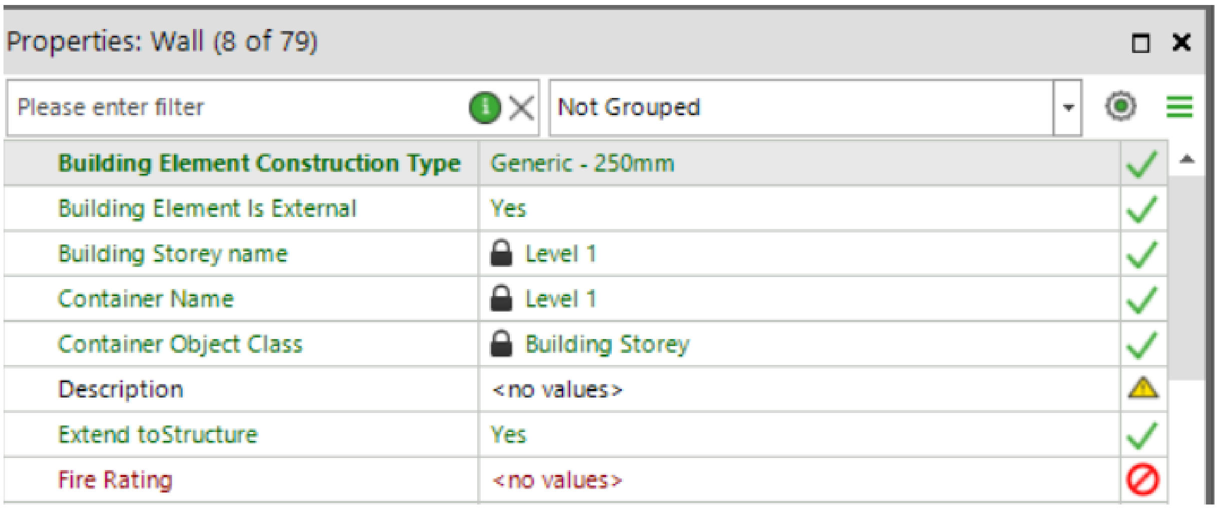
Object [+]	Include	Yes/No Property [+]	Yes/No Value
All	No		
Wall	Yes	Building Element Is External	Yes

Figure 3-8 SimpleBIM Excel template

The same result can also be achieved with a template. First, the objects selected in Figure 3-8 in line 17 are set to zero. Below that, in the next lines, you can now build up which objects are to be contained in the IFC model. Line 18 is intuitively based on the IFC schema. All walls that have the value yes in the Pset_WallCommon within in the property Building Element Is External are selected.

Validation

By setting up a template in SimpleBIM, it is also possible to validate against a specific schema. The property can then be changed directly, or empty properties can be filled.


Properties: Wall (8 of 79)

Please enter filter | Not Grouped

Building Element Construction Type	Generic - 250mm	✓
Building Element Is External	Yes	✓
Building Storey name	🔒 Level 1	✓
Container Name	🔒 Level 1	✓
Container Object Class	🔒 Building Storey	✓
Description	<no values>	⚠
Extend toStructure	Yes	✓
Fire Rating	<no values>	⊘

Figure 3-9 SimpleBIM workspace showing properties of IfcWall

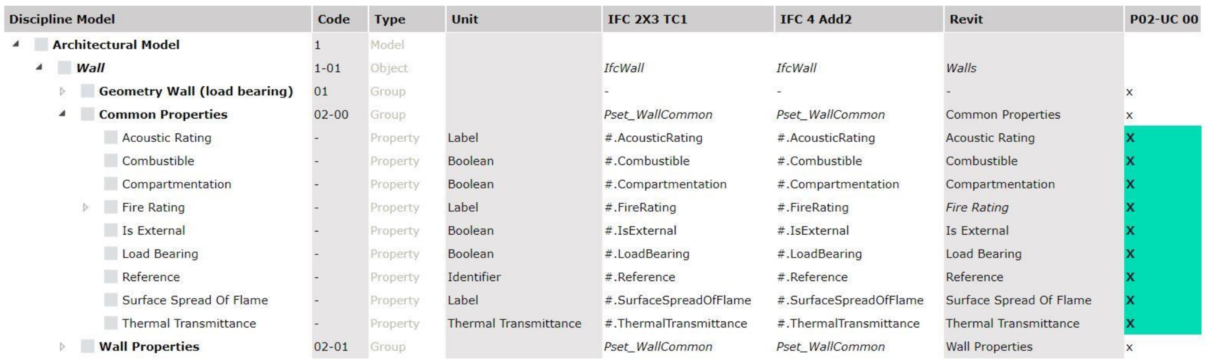
In Figure 3-9 the workspace displays all attributes and their values of all walls that are contained in the model. If an attribute passes the check against the selected Excel template, it will receive a green tick. Attributes that are not contained in the model are shown with a yellow triangle and attributes that contain incorrect or missing values are shown with a red missing sign.

Evaluation of SimpleBIM

SimpleBIM is a well-functioning tool that has made its main task to adapt complex IFC models to individual requirements and thus keep work paths short. One approach that SimpleBIM does not follow in contrast to IfcDoc is to create documentation. Mark-ups and comments on certain parts of a model cannot be achieved with SimpleBIM. Like-wise, full interoperability with open source programs is not yet possible. Although ifc-files can be created, conversion with mvdXML files is still in the development phase. With an add-on, templates can be converted into mvdXML files. The conversion is still subject to some restrictions. For example, no opening elements can be supported. Furthermore, specific applications that are generated within SimpleBIM, such as groups, cannot be mapped in an mvdXML (SimpleBIM, 2020c). Despite the general ease of use, it must be said that a more detailed use requires a great deal of training. For example, the creation of own templates requires a lot of practice and knowledge.

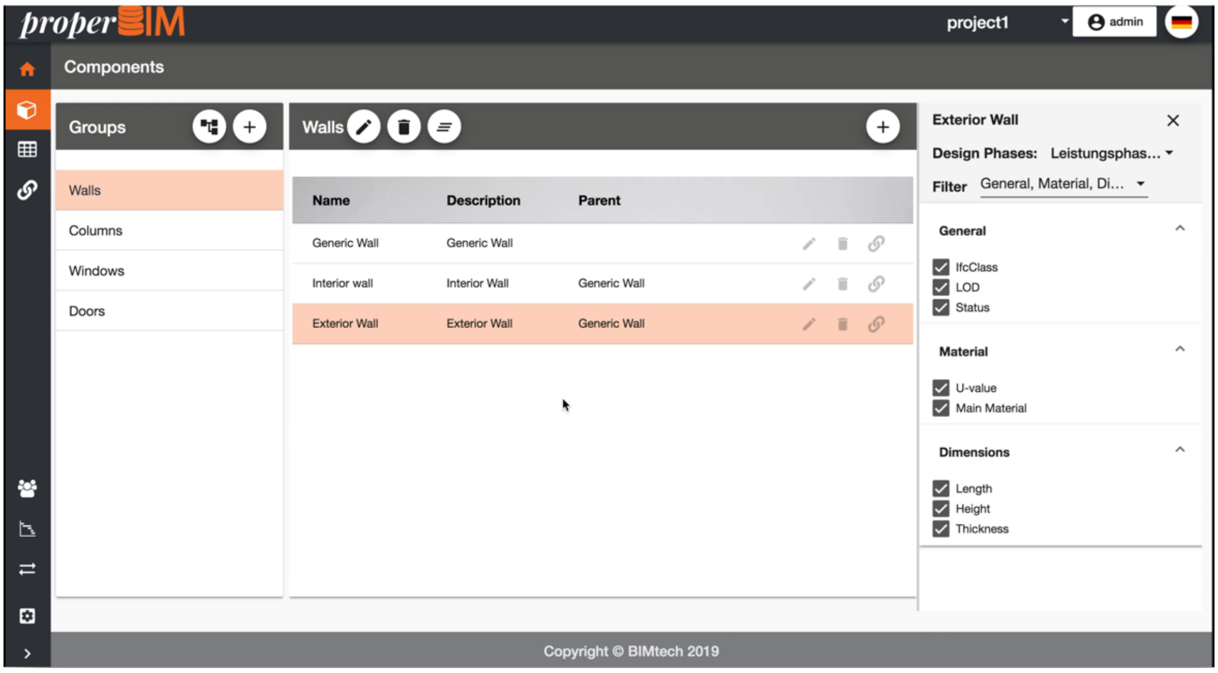
3.4.2 Further commercial Tools

Besides SimpleBIM there are several other commercial programs that deal with the topic of IDMs and MVDs. In general, those web-based platforms offer the user the possibility to create exchange requests based on extensive databases. The main tasks of the applications are the creation and adaptation of ERs and writing of test rules against which an already created model can be validated. These validation rules can then be output as mvdXML files. Even if the integration into other programs via mvdXML tries to keep workflows short, the familiarization with the programs itself can be relatively difficult. Basic knowledge of the IFC schema, as well as of the processes and IDM are necessary for the user. Figure 3-10 and figure 3-11 show workspaces for the two applications BIMQ and properBIM.



Discipline Model	Code	Type	Unit	IFC 2X3 TC1	IFC 4 Add2	Revit	P02-UC 00
Architectural Model	1	Model					
Wall	1-01	Object		*IfcWall*	*IfcWall*	*Walls*	
Geometry Wall (load bearing)	01	Group		-	-	-	x
Common Properties	02-00	Group		*Pset_WallCommon*	*Pset_WallCommon*	Common Properties	x
Acoustic Rating	-	Property	Label	#.AcousticRating	#.AcousticRating	Acoustic Rating	X
Combustible	-	Property	Boolean	#.Combustible	#.Combustible	Combustible	X
Compartmentation	-	Property	Boolean	#.Compartmentation	#.Compartmentation	Compartmentation	X
Fire Rating	-	Property	Label	#.FireRating	#.FireRating	*Fire Rating*	X
Is External	-	Property	Boolean	#.IsExternal	#.IsExternal	Is External	X
Load Bearing	-	Property	Boolean	#.LoadBearing	#.LoadBearing	Load Bearing	X
Reference	-	Property	Identifier	#.Reference	#.Reference	Reference	X
Surface Spread Of Flame	-	Property	Label	#.SurfaceSpreadOfFlame	#.SurfaceSpreadOfFlame	Surface Spread Of Flame	X
Thermal Transmittance	-	Property	Thermal Transmittance	#.ThermalTransmittance	#.ThermalTransmittance	Thermal Transmittance	X
Wall Properties	02-01	Group		*Pset_WallCommon*	*Pset_WallCommon*	Wall Properties	x

Figure 3-10 BIMQs table offering selection of ERs



Name	Description	Parent
Generic Wall	Generic Wall	
Interior wall	Interior Wall	Generic Wall
Exterior Wall	Exterior Wall	Generic Wall

Exterior Wall

Design Phases: Leistungsphas...

Filter General, Material, Di...

General
- ✓ IfcClass
- ✓ LOD
- ✓ Status

Material
- ✓ U-value
- ✓ Main Material

Dimensions
- ✓ Length
- ✓ Height
- ✓ Thickness

Copyright © BIMtech 2019

Figure 3-11 ProperBIM workspace with a selection of external walls

3.5 MVD synopsis

A Model View Definition attempts to limit the very broad IFC schema to a range that is completely sufficient for certain ERs. The necessity for this narrowed view on a specific part of the IFC schema is caused by the IFC schema's motivation for universality. The use of MVDs has changed slightly over time. In the past there was a clear limitation by the official MVDs and their use, the subset specification. However, recent scientific papers indicate that the use of MVDs has extended towards specific use cases (buildingSMART, 2019c). Scientific studies as well as commercial programs like SimpleBIM show that these use cases are frequently related to filtering a building model and automated validation for quality assurance. IfcDoc was developed by buildingSMART to create MVDs. A possible way to satisfy the demand for special use cases within the MVD area is to enable users to create their own MVDs and adapt them to special use cases. But IfcDoc in its current form has major limitations in user-friendliness and important features are not implemented. Especially for non MVD experts it is therefore practically not useable. The fact that more and more commercial tools reflect the functionalities of IDM and MVD shows that MVD is also relevant to the market economy. Commercial tools often have the disadvantage of having been implemented only for precisely tailored tasks. They can also require a high degree of training for particularized tasks. As one of the most widely used commercial tools, SimpleBIM also highlights the importance of integrating geometries. In SimpleBIM, individual elements or entire entities can be selected in order to select or validate them for specific exchange requirements.

By taking all this into consideration, in the context of this work it is tried to conceive an approach which shows the original stringency of MVDs and is specifying the focus on specific use cases. Following the overview of the scientific work, validation and filtering are considered as obligatory for the functioning of an MVD in today's view. The structure of an mvdXML is basically kept. First of all, models are filtered in an applicability part, after which checks can be performed in a validation part. For the programming implementation, mvdXML was not chosen, the two main reasons are the low user-friendliness usability of mvdXML and the restrictions on a geometrical basis. The geometrical limitations are caused by one way linking of entities in separate templates, so no geometrical relations between different entities can be established. In the new approach, geometry should represent an extended possibility to specify information to be exchanged. An imperative language was chosen as the underlying language to provide an easy-to-use entry interface and far-reaching possibilities in the field of geometry. The query language QL4BIM has turned out to be suitable to meet these challenges. QL4BIM is analysed in the following Chapter. The required extensions to QL4BIM are presented in Chapter 5 and EIMC-VP is presented in Chapter 6 as a possibility to facilitate the use of the extended query language.

4 QL4BIM imperative approach for model queries

A completely different approach compared to the descriptive programming is used by QL4BIM. The language was developed to enable typical end user to sufficiently analyse the database of a building model. To achieve this objective, an imperative programming approach was chosen. In imperative programming, a sequence of statements is executed one after the other. The source code explicitly describes the individual steps that the program must perform (Microsoft, 2015). The focus of the programming style is on the 'how'. This approach represents a stringent sequence of commands, which is relatively understandable for humans. The exact description of the solution determines the importance in the sequence of statements. The succession cannot be changed in the imperative approach without changing the result. Within imperative programming, variables are used in the individual steps as containers for data to be passed. These containers do not hold the instance entities of the variable itself, but references that refer to the instance files. The connection of concrete values to a variable is implemented with the assignment command (Wagenknecht, 2016, pp. 175 – 176).

In the context of query languages in BIM it is important to mention that besides QL4BIM there are other open source query languages. These have also been designed with the objective of analysing IFC based models. BIMQL, for example, focuses on the ability to select certain entities of a model and change values of these entities. The query language IfcQL, already mentioned in 3.3.5, was designed to convert mvdXML files to query language code. The main reasons for the decision to use QL4BIM as the query language for this work are the high usability and the possibility to include special geometrical operators in an MVD that allow topologically instinctive queries, like select all doors and all walls that enclose at least one door. The entire Chapter refers to Daum (2018) and gives an overview of the most important aspects within it corresponding to this work. We recommend reading the dissertation for more details.

4.1 Basic principles of QL4BIM

The advantage of the intuitive understanding of an imperative approach in filtering of IFC instance files was exploited by Daum (2018). The result is the imperative query language QL4BIM. A core of QL4BIM is the entity. The entity is specified by an external schema (Daum, 2018, p. 176). Currently the CityGML and the IFC schema are supported. When using the IFC schema, each QL4BIM entity corresponds to an IFC entity. The language specific QL4BIM data type ExternalEntity is used to provide attribute and type information of an entity from the IFC schema (Daum, 2018, p. 168). As the IFC schema is provided to the QL4BIM query runtime, structures of the schema such as types and their attributes can be used in QL4BIM. Although QL4BIM was designed as a general query language, this bachelor thesis will focus on how to use QL4BIM based on the IFC schema. The basic principles of the language are briefly explained.

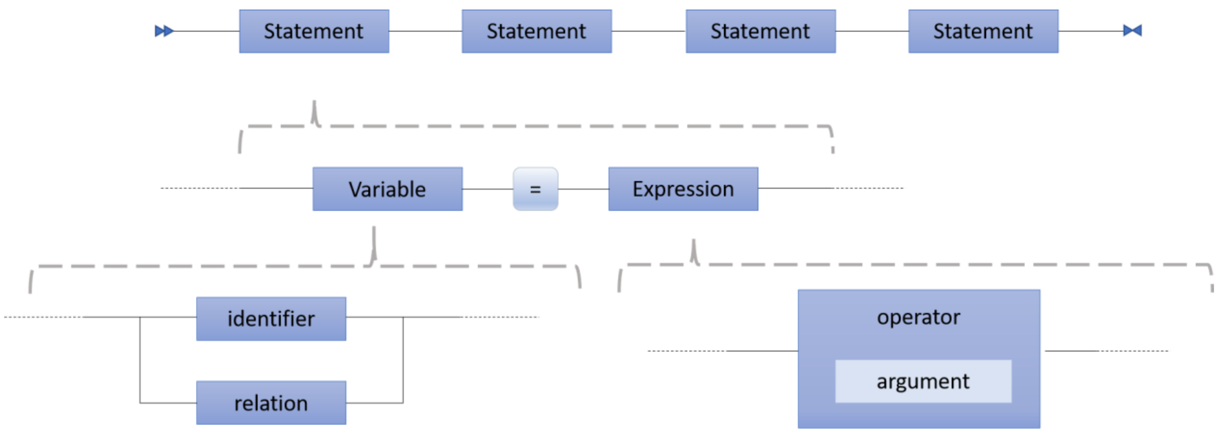


Figure 4-1 QL4BIM conceptual overview (Adopted from Daum, 2018, p. 165)

In the imperative query language QL4BIM, strict sequences of statements represent a query (Figure 4-1). A statement consists of two parts. One part is a variable and the other part is an expression. The variable is thus assigned to the result of the expression and serves as a container for the received data set. The variables are either entity sets or entity relations. Within an expression, the variables serve as arguments of the operators. In addition to the variables, both simple data types such as string or integer and predicates can represent arguments of an expression.

4.2 Variables in QL4BIM

In QL4BIM, variables can represent either entity sets or relations of entities. A set variable contains a list of unordered entities from an instance file (Daum, 2018, p. 171). The technical provision of the entities is realised via GUID. Each individual entity inherited from IfcRoot can be referred to exactly one GUID. Entities of the resource layer have no GUID. Instead, the unique referencing is defined using a local ID.

Wall Entities
IfcWall#100
IfcWall#101
IfcWall#102
IfcWall#103
...

Table 1 Visualisation of a set in QL4BIM with IfcWall instances (Adopted from Daum, 2018, p. 172)

An entity relation represents a set of tuples. A tuple consists of at least two attributes. One attribute represents exactly one column in a relation matrix. One attribute of a relation therefore represents a set of entities. Each attribute is assigned an individual index in the relation. The assignment of indices to the individual attributes of a relation enables the referencing of individual sets that a single relational attribute represents. Instead of the index, the name of the respective attribute can also serve as the declaration of the attribute (Daum, 2018, p. 171). Both the index and the name of the attribute can be used as parameters in subsequent operators to ensure the use of only a certain column of a relation in an operator. However, even if only a certain attribute of the relation is named in an operator, the whole relation is passed into the output variable of this operator, in order to be able to apply possible operations to the previous attributes in the further course.

RelA	
Wall	Storey
IfcWall#101	IfcStorey#3
IfcWall#103	IfcStorey#3
IfcWall#109	IfcStorey#3
IfcWall#371	IfcStorey#4
...	...

Table 2 Visualisation of a relation. RelA representing the relation between IfcWall and IfcStorey entities (Adopted from Daum, 2018, p. 172)

A set in QL4BIM always contains complete entities and no simple types, just as a relation contains tuples of complete entities. Simple data types such as integer, string or float values are stored as constants (Adopted from Daum, 2018, p. 176).

4.3 Operators

The previously defined variables can be the result of an expression and used as input parameters of an operator. The operators analyse the BIM models. An excerpt of operators in QL4BIM is shown in Table 3 and Table 4. Since a variable can be either a set or a relation of entities, QL4BIM is implemented so that an operator follows different underlying pattern depending on the input type of the variable. This multiple usage of the same operator is called overloading and allows the end user to easily handle various data processing, since he does not have to pay attention to what type a variable is (Daum, 2018, p. 196).

Operator	Alias	Description
ImportModel	IM	Import a IFC Instance-file
ExportModel	EM	Export a IFC Instance-file
TypeFilter	TF	Type Selection
AttributeFilter	AF	Attribute Selection
Projector	PJ	Projection of relational variables
Dereferencer	DR	Resolution of attributive reference relationships
Union	UN	Union of quantities or relations
Difference	DIF	Difference of quantities or relations
Contains	CT	Topologically Contained
Inside	IN	Topologically inside
Touches	TO	Topological contact
Nearer	NE	closer than a limit value

Table 3 Important QL4BIM elementary operators for this work (Adopted from Daum, 2018, p. 196)

The operators in Table 3 are used to process entities. The function of these operators cannot be replicated by other operators. In addition to these elementary operators, further operators have been developed to make the examination more effective. The following operators are examples of this group. They are called secondary operators and are distinguished by an internal use of other QL4BIM operators. The most important ones for this work are represented in the following Table 4.

Operator	Alias	Description
PropertyFilter	PF	Quantity of properties
Deassociator	DA	Dereferencing of inverse relationships
Merger	ME	Three way merger with geometry analysis

Table 4 Important QL4BIM secondary operators for this work (Adopted from Daum, 2018, p. 198)

The most important operators in QL4BIM for the new approach are examined in more detail in the following sections.

4.3.1 TypeFilter

The TypeFilter-operator selects entities of a certain class from a set or relation of entities provided by the input parameters. By aggregating the IFC schema, the hierarchy of the schema can be used and subclasses of the selected class are automatically included in the result (Daum, 2018, p. 200). The following two lines illustrate the use of the TypeFilter when an entity set is used as input parameter.

```
1   entities = ImportModel ("C:\ Testmodel .ifc ")
2   walls = TypeFilter ( entities is IfcWall )
```

Listing 4-1 Usage of the TypeFilter operator in QL4BIM (Adopted from Daum, 2018, p. 201)

The first line returns in Listing 4-1 all entities of the IFC model with the variable entities. The variable initialized by this statement can now be passed as argument to the Type-Filter. The operator filters all entities of the passed argument to the set contained in the predicate. In this example the predicate 'is IfcWall' initiates the selection of IfcWall instances and their subclasses, such as IfcWallStandardCase. If a relation is passed to the operator as an argument instead of a set, each attribute of the input tuple can be filtered individually for a type.

```
1   entities = ImportModel ("C:\ Testmodel .ifc ")
2   a[ whole | part ] = Deassociater ( entities . Decomposes )
3   a[ roof | slab ] = TypeFilter ( a, [ whole ] is IfcRoof , [ part ] is IfcSlab )
```

Listing 4-2 Using the TypeFilter operator in QL4BIM with a relation as data source (Adopted from Daum, 2018, p. 201)

As in the first example, all entities contained in the model are stored in the variable entities. In line 2, these are now resolved referentially by the Deassociator operator with the Decomposes attribute. The objective relationship is resolved by means of a double dereferencing. The output of line 2 with the variable a[whole | part] is a relation, which contains whole and parts in its indices, as results of the decomposition. The TypeFilter operator is now applied to this relation. While on the first index whole is filtered to IfcRoof, on the second index a filtering to IfcSlab is performed. As a result, the variable a[roof | slab] in line 3 contains all entities of the class and subclasses of IfcRoof in the first relational attribute. In the second relational attribute all entities of the class and subclasses of IfcSlab are stored.

4.3.2 AttributeFilter

The AttributeFilter provides another important filter option within the IFC schema. The operator filters entities with certain attributes. These attributes can be further restricted directly within the argument to strings, specific values, or user-defined limits for a value (Daum, 2018, p. 202).

```
entities = ImportModel ("C:\ Testmodel.ifc ")
doors = TypeFilter ( entities is IfcDoor )
highDoors = AttributeFilter ( doors . OverallHeight >= 2.0)
```

Listing 4-3 Using the AttributeFilter operator in QL4BIM (Adopted from Daum, 2018, p. 202)

After importing all entities, a selection of all doors is made with the TypeFilter operator already introduced. Afterwards this filtering is limited to doors which are larger than 2.4 units. This attribute specific filtering is done by the AttributeFilter operator. Within the argument of the operator the variable is named first, then the attribute name is declared, separated by the dot operator. Using a comparison operator and then determining the comparison value makes it possible to restrict the set of entities to those that satisfy the attribute predicate. The next example shows the use of the AttributeFilter with a relation as input variable.

```
1   entities = ImportModel ("C:\ Testmodel.ifc ")
2   walls = TypeFilter ( entities is IfcWall )
3   doors = TypeFilter ( entities is IfcDoor )
4   a[ wall | door ] = Touches (walls , doors )
5   b[ wall | door ] = AttributeFilter (a, [ Wall ]. OverallWidth >= 2400)
```

Listing 4-4 Using the AttributeFilter operator in QL4BIM with a relation as data source (Adopted from Daum, 2018, p. 203)

After the entities of IfcWall and IfcDoor have been stored in the variables walls and doors in line 1 to 3, a relation is created by using the Touch operator, which contains tuples in which touching IfcWall and IfcDoor instances are located. The variable a in the above example is separated and afterwards the declaration, which attribute of the relation should be selected to be filtered, is done. The following restriction of the entities is done as in the use with Set variables by a delimitation with the dot operator and naming of the attribute and the value range to be restricted.

4.3.3 PropertyFilter

The PropertyFilter allows to select certain property sets and properties within them. While the AttributeFilter examines direct properties of an entity, the PropertyFilter operator analyses IfcPropertySets associated with the entity. The property sets can be linked to the entity either via occurrence or via the entity type. The implementation of the PropertyFilter simplifies the representation. In the implementation the relationships of the property set are dereferenced, and all received properties are compared with the desired name of the property. If a match occurs during the comparison, the desired property is included in the result set and linked to the entity again by means of the objectified relationship (Daum, 2018, p. 232).

```
entities = ImportModel ("C:\ Testmodel.ifc ")
walls = TypeFilter ( entities is IfcWalls )
externalWalls = PropertyFilter ( walls.isexternal = true )
```

Listing 4-5 Using the PropertyFilter operator in QL4BIM with sets

In the example above, all entities of the model that belong to IfcWall are checked to see if they have the property IsExternal in a linked property set. The value of this property must also have the value TRUE. The operator can be extended, so that the name of a property set must also satisfy a certain user-defined string. In this case, only property sets with the corresponding name are examined and the properties contained in them are subjected to the comparison with the predicate in the argument of the operator. The following example illustrates this case.

```
1 entities = ImportModel ("C:\ Testmodel.ifc ")
2 elementProxy = TypeFilter ( entities is IfcBuildingElementProxy )
3 externalWalls = PropertyFilter (elementProxy.Classification=slab, “Pset_Identification”)
```

Listing 4-6 Using the PropertyFilter operator to examine properties in specific property sets

Only the property sets with the name Pset_Identification are examined. The filtering only focuses on the property classification within this property set. Furthermore, the property must satisfy the string value slab.

The PropertyFilter is a secondary operator that contains a combination of the functions of the Dereference-operator and the AttributeFilter. The signature within the argument of the PropertyFilter is analogous to the AttributeFilter. Properties as well as property sets can be filtered. In order that the property sets or properties linked via relationships can be processed, this relationship must be dissolved. This is done, as mentioned above, by means of dereferencing. The Dereferencer represents this dereferencing as an elementary operator in a detached way.

4.4 QL4BIM conclusion

Features of QL4BIM include that it can be used by domain users with minimal programming knowledge, that it conforms to widely used high-level languages such as Java, C# or C#++ and that input data are open data models whose schema is mapped and remains unchanged.

These features make QL4BIM suitable as component for MVD bases filtering and validating of IFC models. Therefore, some conceptual enhancements and practical implementations of QL4BIM are developed for this thesis. The newly developed conceptually approach is called Extended Imperative Model Checking. It uses QL4BIM as target language and combines it on the one hand with the logical structure of an mvdXML and on the other hand with the expressiveness of the visual programming language Blockly (2020a). In the following Chapters the approach and a created prototype will be discussed.

5 Extended Imperative Model Checking

An MVD is created for manifold purposes. It can be used to select a specific sub-schema of an IFC model, to document the creation and use of this sub-set, to filter data, and to check whether a set of data meets certain constraints of an Exchange Requirement (a more detailed view on purposes of an MVD is provided in section 3.2). MvdXML tries to map all these requirements in a data model. XML provides the appropriate base language for this with its extension and customization possibilities. MvdXML is customized to map the complex IFC data model correctly and to perform the various tasks. For example, a simple property set that is to have a value of the property heat transfer can be assigned to an entity in different ways. Either it can be assigned to occurrence (e.g.: IfcWall) or to type (e.g.: IfcWallType) whereas the later approach is usually used to map the property set to many occurrences. The large number of use cases in combination with the fine grained and highly referential IFC model leads to a complex process when creating and processing an mvdXML file.

This resulted in the idea of a new approach for creating MVDs for specific use cases. An elementary important aspect of the new approach is usability. Therefore, in this thesis the focus is limited and does not cover all possible uses of an MVD. The all-inclusive attempt would inevitably lead to a loss of handiness. The incorporation of automated validation in ‘mvdXML 1.1 specification’ shows that validation is a required feature of an MVD. The subsequent development of Zhang's and colleagues’ (2015) MVDChecker also implies the importance of validation and its execution. Thus, the provision of an automated validation facility in the context of MVD is considered mandatory. Next to the validation, filtering is seen as another important feature of MVDs. The name of the new approach is Extended Imperative Model Checking (EIMC). It already contains important basic properties in its name. The term *Extended* is used to indicate expanding geometry operation options. *Imperative* describes the underlying programming paradigm. *Model* refers to the execution directly on the model on instance level. And *Checking* represents with filtering, one of the two main functions. The query language QL4BIM is used for the new approach as a basis. The reasons for this decision included the intuitive use and that filter operations can already be performed on ifc-files. The query language will be extended in such a way that automated validation will also be possible. The sequential structure of the query language will be separated into an Applicability and a Validation part. Furthermore, the user-friendliness will be increased by a visual programming application EIMC-VP based on the concept of EIMC. The application aims to enable the end user to create an MVD in a simple way.

The above described purposes of the new MVD generator lead to the following elementary claims regarding the underlying concept of EIMC. The first part of requirements and settings is congruent to the requirements of QL4BIM (Daum, 2018, p.164), further requirements are available in the second part.

Part 1:

- The language should have a high degree of usability. Domain user with little or no programming skills should be able to understand the usage of the resulting query language
- The language should be able to handle common BIM formats. The input and output of '.ifc' files are of particular importance. Furthermore, the entities of these models shall remain unchanged.
- EIMC shall have conformity to high-level languages such as Java, C#, C++, Python.

Part 2:

- The language is designed to be able to reproduce basic functions of a standard MVD as produced by mvdXML. Therefore, parts of the language of QL4BIM are not represented within EIMC and validation operators are added.
- A visual programming language is to be based on the EIMC concept. The representation by means of visual programming should make the functionality intuitively accessible to the user.

5.1 EIMC Concept

The two main aspects to filter and to validate define the structure of concept of EIMC. Based on the structure of the mvdXML format, firstly the filtering is done. Secondly a validation of the previously performed filtering is enabled by validation operators. The two functional fields of the language are to be considered separately and can be executed independently (although there is always a pre-filtering on the entities to applicable type due to the ImportModel operator).

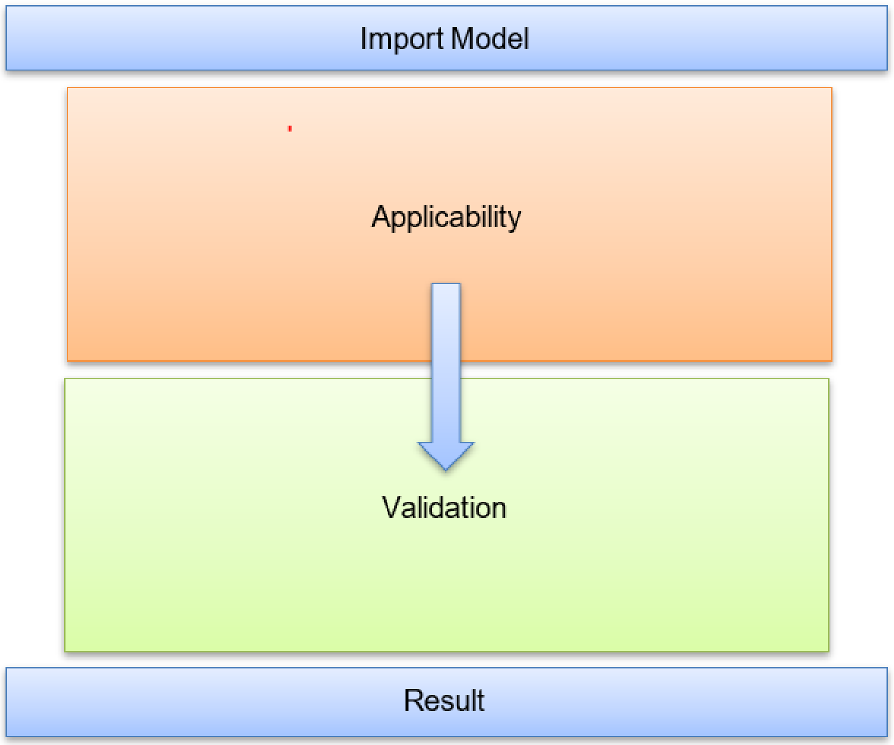


Figure 5-1 Schema of EIMC

Figure 5-1 shows the conceptually structure of EIMC in a simplified way. The assembling starts with the opportunity to import an IFC instance file. The filtering of the entire IFC model is then carried out in the Applicability part. The selection is done by filter operators and can be refined by many more operators. For example, operators can be used to display various topological relationships between entities. Subgraphs of the IFC schema can be easily created by selecting hierarchically superior entities. The selected entities are stored in the individual variables via references. Subsequently, the validation of the selection specified in the Applicability part provides the check of selected entities. In the Validation variables can be checked for various conditions and restrictions. The scope and function of the check operators are presented in detail in the next section. The application of filtering and validation on an IFC model is the output in the result area. Since the two parts Applicability and Validation can be used independently, the output can be a filtered IFC model as well as a check report containing a validation review.

5.2 Validation Operators

This section introduces the new validation operators and explains how they work. In total, three new operators will be introduced. The AttributeCheck, The PropertyCheck and the more general XCheck. The operators allow to validate subgraphs of the IFC model against constraints. Within the sub-graphs attributes as well as properties can be validated. According to the metric of mvdXML in both cases the existence check of attributes or properties as well as the check of values of attributes or properties are of major importance. While the ExistCheck only checks the presence of attributes or properties, the ValueCheck allows you to check simple data type queries, for example if the property name is equal to a certain sequence of strings. To simplify user handling as much as possible, the XCheck operator was developed. It contains all applications of the two validation operators AttributeCheck and PropertyCheck. Depending on the input, the XCheck operator adapts its execution. If either explicitly only attributes or explicitly only properties should be checked, it is to use the AtttributeCheck or PropertyCheck. If properties have the same name as attributes, using the XCheck will cause both to be checked.

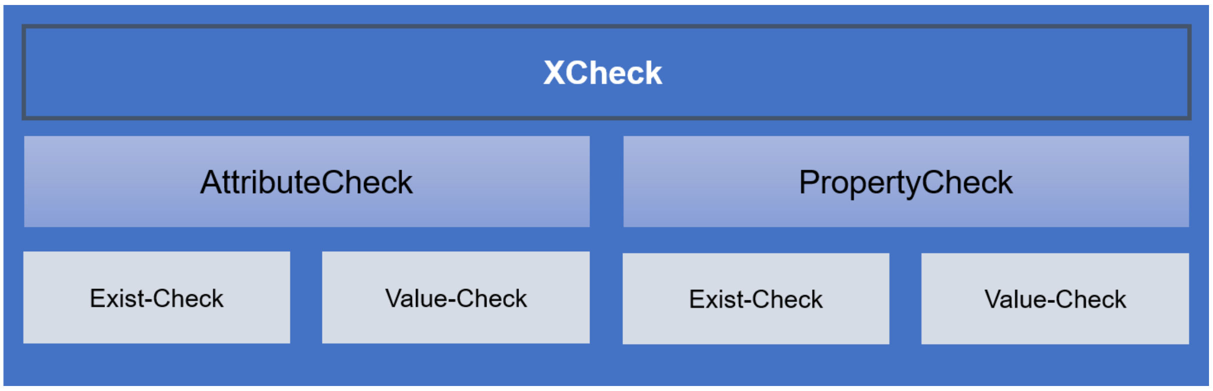
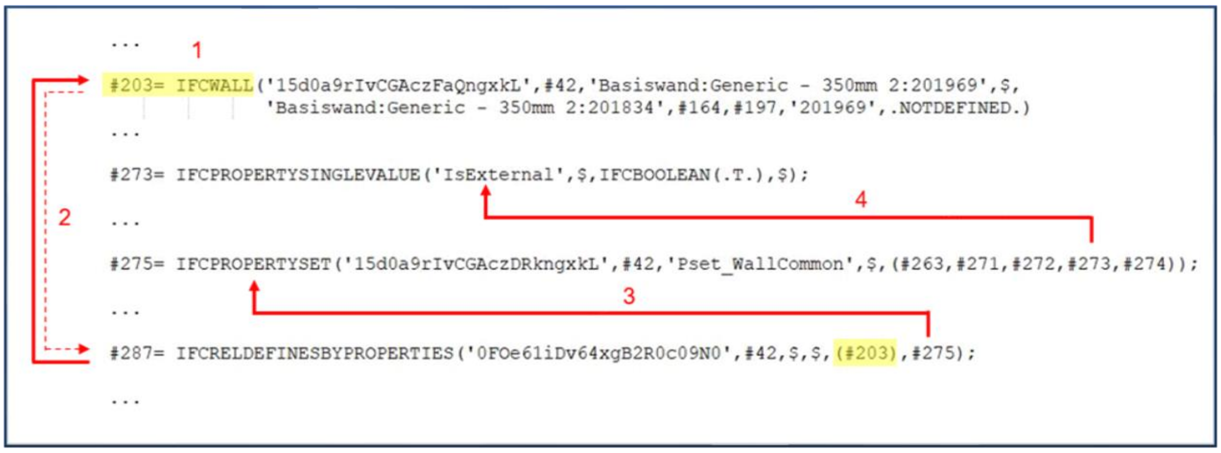


Figure 5-2 Classification of the new validation operators

5.2.1 General functioning of validation operators

The validation operators perform a comparison of their argument with the instance file. They search the IFC instance file either for attributes of the desired file or for property sets that are linked by objectified relationships. If the algorithm finds the corresponding property in the IFC instance file, it checks whether the restriction determined by the argument of the validation operator matches the content of the instance model. If this is the case, the validation is immediately declared as passed in a report file. The report file is used to save results instantly and is used as an error review. If the comparison does not match, the validation is declared as failed. In the case of an attribute validation, only the corresponding entity (represented as one line in the STEP-21 file) needs to be searched in order to perform the comparison. Since the attributes are declared directly in the same line where the element is described. If a property set is to be checked, the entire STEP-21 file is first searched for property set linking options. Figure 5-3 shows an example of a comparison that checks whether elements of the entity IfcWall have assigned the property IsExternal. It is analysed whether the property is present or not. The links presented in 2.2 must be checked. The algorithm searches the file for IFCRELDEFINESBYPROPERTIES, which has references to the instances of IfcWall as well as to corresponding property sets.



```
...
#203= IFCWALL('15d0a9rIvCGAczFaQngxkL',#42,'Basiswand:Generic - 350mm 2:201969',$,
                'Basiswand:Generic - 350mm 2:201834',#164,#197,'201969',.NOTDEFINED.)
...
#273= IFCPROPERTYSINGLEVALUE('IsExternal',$,IFCBOOLEAN(.T.),$);
...
#275= IFCPROPERTYSET('15d0a9rIvCGAczDRkngxkL',#42,'Pset_WallCommon',$,(#263,#271,#272,#273,#274));
...
#287= IFCRELDEFINESBYPROPERTIES('0FOe61iDv64xgB2R0c09N0',#42,$,$,(#203),#275);
...
```

Figure 5-3 Explained execution of the check function on an instance file

The property set contains individual references to the linked properties. Therefore, the algorithm does not scan the file from top to bottom one time but jumps back and forth when links are found. When the algorithm has found a fitting link, it writes the result into the check report. In Figure 5-3 the algorithm would note that element #203 passed the check since the property IsExternal is part of one of the property sets linked to it.

5.2.2 Validation operator XCheck

In the XCheck operator, both property and attribute check operator functions are conceptually cumulated. Therefore, only the XCheck is described in detail. The notation of the first overloading of the XCheck operator is shown in Listing 5-1.

```
1  entities = ImportModel ( “C:\Testmodel.ifc” )
2  wall = Typefilter ( entities is IfcWall )
3  a = XCheck ( wall. Description )
```

Listing 5-1 Using the XCheck operator in EIMC

Listing 5-1 examines the XCheck whether the elements of the entity IfcWall have assigned an attribute or property named Description. After having imported the IFC model in line 1 and limited the set of all entities to IfcWall in line 2, the operator executes in line 3. First it performs the comparison with the attributes and then a comparison for properties. The operator checks the instance file and writes simultaneously into a JSON file, which elements of the entity IfcWall had an attribute or property called Description and which elements not. It is also possible to further limit the scope of the operator. With the following overload the XCheck can be used to check only properties in certain property sets.

```
entities = ImportModel ( “C:\Testmodel.ifc” )
wall = Typefilter ( entities is IfcWall )
a = XCheck ( wall. IsExternal, “PSet_WallCommon” )
```

Listing 5-2 Using the XCheck operator with selection of a special property set

In Listing 5-2 it is checked whether each element of IfcWall has a property called IsExternal. Only properties in the property set Pset_WallCommon are checked. Therefore, the property must be in exactly this property set, if a property in another named property set complies with the rules it will not be checked.

Instead of sets, relations can also serve as input argument of the XCheck operator. A relation consists of several columns of different sets or relations. A check can be specified for only one column, for several columns and as shown in the next example also for different checks for different columns.

```
1  entities = ImportModel ("C:\ Testmodel .ifc ")
2  a[ whole | part ] = Deassociater ( entities . Decomposes )
3  a[ roof | slab ] = TypeFilter ( a, [ whole ] is IfcRoof , [ part ] is IfcSlab )
4  b = XCheck ( a, [roof].Description, [slab].Name )
```

Listing 5-3 Using the XCheck operator in EIMC with variable as data source

Listing 5-3 establishes in lines 1-3 a relation. The relation consists of the two entities IfcRoof and IfcSlab. In the check operator single columns of the relation can be examined. Each row is examined individually. For all elements of IfcRoof it is checked whether an attribute with the name Description is present. For all elements of IfcSlab, however, it is checked whether an attribute Name is present. The notation of the results in the JSON file is done separate from each other.

If in the argument of the XCheck operator a comparison operator (<,>,=) implies another value, the operator not only allows to check the existence of an attribute or property but also to compare its value to the given predicate.

```
entities = ImportModel ( “C:\Testmodel.ifc” )
wall = Typefilter ( entities is IfcWall )
a = XCheck ( wall. Description = “TestDescription” )
```

Listing 5-4 Using the XCheck operator with the specification of a special value

Analogous to the examples before, initially a file is imported and a filtering on IfcWall elements is carried out. The operator then searches the elements from IfcWall for attributes and properties of the name Description. If these are found, it is additionally checked whether the value of the corresponding property matches the string 'TestDescription'. If the match is successful, a note is attached to the JSON file.

It is also possible to let the operator check the instance file only for certain property sets. Additionally, it is possible to define integer and floating values in the argument. In contrast to the string-based value comparison, an integer or floating value can also have a certain limit. This limitation is initiated with a greater/smaller character and has the consequence that all values which are greater or smaller than this limit pass the validation.

```
entities = ImportModel ( “C:\Testmodel.ifc” )
wall = Typefilter ( entities is IfcWall )
a = XCheck ( wall. ThermalTrancemittance < 1.3, “PSet_WallCommon” )
```

Listing 5-5 Using the XCheck operator with a limit for a specific property in a declared property set

As well as the string-based comparison, the property is written behind the variable separated by a dot-operator. Then a lower-case character indicates a range of values that is sufficient to fulfil the check. In Listing 5-5 all ThermalTrancemittance properties with value assignment less than 1.3 are considered sufficient and pass the check. Only properties in the property set Pset_WallCommon are considered.

In the XCheck operator, input variables can be relations, then each row is treated individually.

```
1   entities = ImportModel ("C:\ Testmodel .ifc ")
2   a[ whole | part ] = Deassociater ( entities . Decomposes )
3   a[ roof | slab ] = TypeFilter ( a, [ whole ] is IfcRoof , [ part ] is IfcSlab )
4   b = XCheck ( a, [roof].Description = “TestDescription”,
5                        [slab].ThermalTrancemittance < 1.3 )
```

Listing 5-6 Using the XCheck operator with limited variables as data source

Again, lines 1-3 of Listing 5-6 build up the relation consisting of roofs and slabs. In the XCheckValue operator, in addition to just entering the attributes or properties, a restriction to the values of these can be made. In line 4 the section ‘a[roof].Description = TestDescription’ postulates that all elements of the entity IfcRoof in the attribute Description must have the value ‘TestDescription’ in order to pass the check. In line 5 the elements of IfcSlab are checked to ensure that a property ThermalTrancemittance with a value smaller than 1.3 must be assigned to the corresponding elements.

5.3 Use case

The explanation of the developed data model and the new validation operators is carried out in this section using a complete example. The use case is based on the validation example from the mvdXML specification 1.1. In section 3.3.3 an excerpt of this example is already executed in mvdXML format and in the appendix an illustration of the complete textual representation of the mvdXML is given. Furthermore, the request is extended with a topological condition to represent the possible usage of geometric operators. The whole query is defined by providing a property set called Pset_WallCommon with the properties FireRating and ThermalTransemittance for all load bearing and external walls. Furthermore, the selected walls must have the attribute PredefinedType. The request is extended, so that all doors and windows within the selected walls are additionally checked whether they also have the properties FireRating and Thermal-Transemittance.

```
 1    entities = Import Model („C:\Testmodel123.ifc“)

 2    walls= TypeFilter ( entities is IfcWall )
 3    bearingWalls = PropertyFilter ( walls.LoadBearing = True, „Pset_WallCommon“ )
 4    ebWalls = PropertyFilter ( bearingWalls.IsExternal = True, „Pset_WallCommon“ )
 5    windows = TypeFilter ( entities is IfcWindow )
 6    doors = TypeFilter ( entities is IfcDoor )
 7    WindowDoor = Union ( windows, doors )
 8    applicableObjects [ ebWalls | WindowDoors ] = Inside ( ebWalls , WindowDoor )

 9    check1 = PropertyCheck    ( applicableObjetcs, [ebWalls].firerating,
10                               [WindowDoors].firerating )
11    check2 = XCheck           ( applicableObjects,[ebWalls]. ThermalTrancemittance,
12                               [WindowsDoors].ThermalTrancemittance )
13    check3 = AttributeCheck   ( beWalls.predefinedtype )
```

Listing 5-7 Example of a complete query in EIMC

In line 1 of Listing 5-7 all entities of the model are stored in the variable entities. The provision of the file path within the argument of the operator manages the assignment of a corresponding instance file. Thereupon in line 2, the TypeFilter operator selects all entities of IfcWall including all subclasses. Based on this filtering, the PropertyFilter operator is used to filter out only those entities from all walls which have a property loadbearing with the value TRUE. At the end of the argument, the mention of the property set name also specifies that only a linked property set with exactly this name is examined. The possibilities that the property set Pset_WallCommon, in which the examined property is embedded, is directly linked to the occurrence via the objectified relationship IfcRelDefinesByProperties as well as the link via the class IfcTypeObject with the objectified relationship IfcRelDefinesByType are automatically considered. Should both be the case, the closer connection is always used, respectively the association to occurrence. The previously filtered walls are filtered one more time in line 4 using the same method as in line 3. This time the output variable ebWalls contains only the walls that are additionally declared as exterior walls. Line 5 and 6 create two completely new variables. One of these variables contains all elements of the entity IfcWindow and the other contains all elements of IfcDoor. From the three created variables - ebWalls, windows and doors - a bivalent variable is created in line 8. The Inside operator adds all windows and doors to the relation which are located inside one of the loadbearing external walls. Due to the fact that the Inside operator only permits two different variables in its argument, the two sets doors and windows have been merged to one set with the Union operator in line 7. The output variable WindowDoor contains a list in which all entities of the variable doors are listed first, followed by all entities of the variable windows.

In the check, the filtered entities can be used to query validations. In line 9 - 11 the first validation is performed. The PropertyCheck operator checks whether the property specified in the argument exists for the variable. Since there is no comparison operator after the property name, the operator knows to execute the existence check and not a value comparison. The variable applicableObjects in the argument represents a double relation of a set of external, loadbearing walls and a set of windows and doors. Since both properties should be checked if they have attached a property FireRating, both columns are declared in the argument. First the set of loadbearing and external walls is checked for the presence of the property FireRating, then the same is done for the elements in WindowsDoors. Results are noted in the check report. The same procedure is then performed on line 10 with the property ThermalTrancemittance. Here a XCheck operator was chosen to show that the same result is produced. The difference is that first all attributes are examined for the chosen name, secondly all properties. Passed or Failed results are written from the algorithm again instantaneously to a JSON file. Since only the walls should be checked if they have attached a `PredefinedType`, the third check in line 13 is just applied to elements of the set ebWalls. Also, the results are written directly into the JSON file.

5.4 Comparison of mvdXML and EIMC

Due to the user-friendliness and thus deliberate restriction of EIMC in various purposes and the employing of the imperative query language QL4BIM, there are advantages compared to an MVD created in mvdXML format. Likewise, the restrictions also lead to a number of functions that can no longer be performed or can only be performed to a different extent. These aspects are discussed in this section.

Within the mvdXML schema the reusability of templates and concepts is of great importance due to a structure that can be used in many different ways. This leads to the fact that for a domain expert who has knowledge of IFC schema but no programming skills, it is almost impossible to understand the basic schema of an MVD created with mvdXML. In contrast, EIMC, with its imperative and clear style from QL4BIM, provides an easy entry point for domain experts. The Application EIMC-VP aims to simplify the usage even more and tries to give an entry point for domain experts with no programming skills.

The complexity of the mvdXML schema inevitably means that the creation of new MVDs as well as the extension of existing MVDs is hardly possible for non mvdXML-experts. Even official tools like IfcDoc are difficult to use for domain experts due to lack of functionality and a high level of required knowledge about the IFC schema and can only partially solve this problem. EIMC and the resulting visual programming application EIMC-VP, which will be discussed in the following section, attempts to simplify the creation of features of an MVD.

Within a mvdXML rule only one path between entities can be executed. Therefore, only a set selection is available and geometrical queries that refer to multiple objects of different classes are not possible. Within EIMC queries can be created due to geometrical operators, which allow a wider range of geometric queries.

Relationships within the IFC schema can link entities in a variety of ways. For example, a property set can be attached to a class via both occurrence and type. The mvdXML schema requires the specification of all possible relationships to determine whether certain entities are linked to other entities. In the EIMC all possible mappings are searched by using operators and you do not have to enter all possible mappings of a property set manually, due to the implementations in QL4BIM.

QL4BIM currently only supports the BRep display format. Therefore EIMC also just supports BRep as display format. If other or additional geometric representations are available in an ifc-file, QL4BIM converts or specifies itself to the geometric representation by means of BRep.

6 EIMC-VP: A visual programming layer for EIMC

EIMC is developed to allow domain users to apply MVD concepts to instance models as easily as possible. Therefore, the implementation of a visual programming language and the execution of the language were already considered during the development of the concept. The visual programming language should make it even easier for the user to understand and use the query language and subsequently apply an MVD on an instance model. At first a short text will state briefly general advantages and disadvantages of visual programming, whereupon the developed visual programming concept will be presented with examples.

6.1 Visual Programming

Visual programming uses graphic elements to create programming code. These graphic elements often combine several textual commands of the written code into a single graphic element – node (in the following, instead of the term node, the term block is used, which is more appropriate for the chosen visual programming matter). This makes it easier for a user of the programming language to use it. Likewise, the exact syntax of individual commands and their sequence is specified for the user and does not have to be observed by them as the user cannot disregard the syntax due to specifications of the blocks. On the other hand, this combination of different commands into a single graphic element also limits the possibility of use. Visual programming languages are often less versatile in their use or modification (Ritter et al., 2015). Within EIMC, the requirements for a program are limited compared to general programming, so visual programming is considered a good choice in this area.

When talking about visual programming in the context of BIM, one of the most common applications is Dynamo. Visual programming with Dynamo has been developed to make it easier for the end user to create own geometrical representations in Revit. Even though Dynamo can work in its current version independently, and therefore without the Revit environment, the initial idea of parametric modelling of geometries is still very much in evidence (Autodesk, 2020). In this thesis it is tried to enable the creation of an MVD. Settings have to be made, which parts of a model should be included in the MVD and to what extent they should be checked. For visual representation of this kind of purpose, other VPs are better suited. In order to be as close as possible to the main features of EIMC, it was decided to develop a new visual programming language, called EIMC-VP. The visual programming language was developed on the basis of Blockly since it already provides elementary basic functions when creating a new VP language. This means that a fully functional user interface can already be adopted, new blocks and their behaviour can be implemented in a variety of ways and a code generator can be implemented for these new blocks. Blockly (2020a) states that

> From a user's perspective, Blockly is an intuitive, visual way to build code. From a developer's perspective, Blockly is a ready-made UI for creating a visual language that emits syntactically correct user-generated code. Blockly can export blocks to many programming languages, including these popular options: [JavaScript, Python, PHP, Lua, Dart.] (Blockly, 2020a)

The quote clarifies how and for what Blockly can be used. To implement the prototype in this work we used the already existing Blockly environment to create EIMC-VP. The user interface was adapted, the used blocks were implemented and for these custom designed blocks code generators were implemented, which output an intermediate code (further information on the implementation can be found in section 7.2). Blockly is developed by Google and provided as open source.

6.2 EIMC-VP Concept

EIMC-VP is an approach to simplify the creation of EIMC code through visual programming. EIMC-VP enables users to generate a syntactically correct query without having to consider the exact syntax of the language. Furthermore, the user interface is used to execute the code directly on an IFC instance model. Therefore, a backend is set up, which interprets the code created in EIMC-VP and executes it on the selected instance file. The concept of EIMC-VP represents some simplifications compared to EIMC, which allows the user to create a desired MVD easily and quickly. An elementary factor of EIMC-VP is hiding variables of EIMC. The decision not to display variables in the user interface is due to the simple reason that using them would make the application more complex than necessary. To enable users to make the most of the options of EIMC's textual representation, extensions have been built into the visual concept of EIMC, which can be used additionally depending on the requirements of the application. The conceptual structure of EIMC-VP is first described schematically in this section, and then the actual application of the concept is shown.

Since no variables should be visible to the user, processes must be defined automatically. A simple sequence of three statements in a row is used in the visual display in such a way that the output variable of the previous statement is always used as the input variable of the next field. A refinement of selection results, which are considered elementary for the purposes of filtering and validation, is thus considered. The first input variable always contains all entities of the entire model.

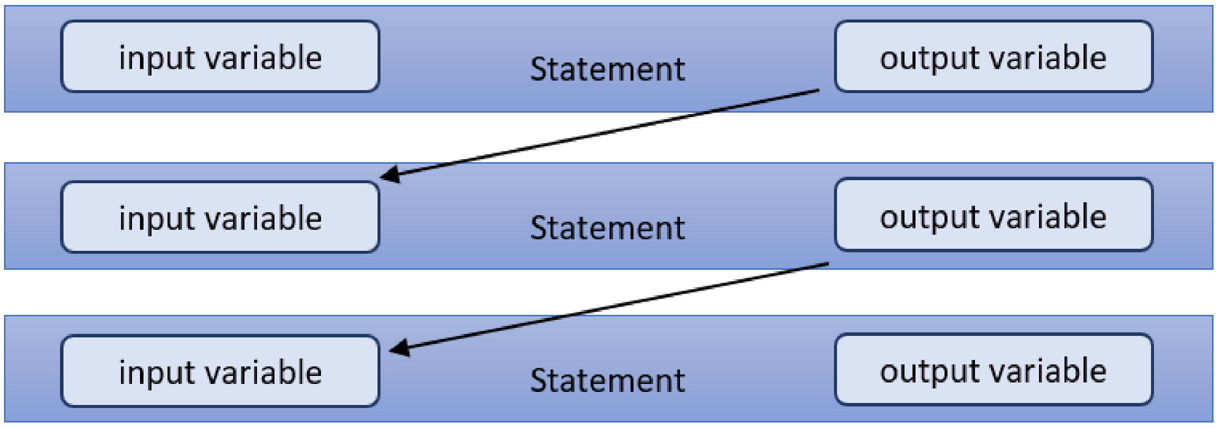


Figure 6-1 Automated variables passing on statements which are arranged below each other

The basic structure of the visual representation resembles the textual representation. In the first part filters are applied to the IFC model. In the second part, validations of the IFC model are performed. In the applicability part the direct transfer of variables can be interrupted. Several applicability parts can be set up separately (In figure 6-2 the individual applicability parts are shown schematically separated with an AND-block). These are independent of each other and are all checked independently in each check. Every applicability part starts in the first statement with an input variable, which contains all entities of a possible instance file. The output variable of the last statement of a direct sequence of statements is then checked in the validation part by validation operators. Without further restrictions by the user each of these output variables is then checked for each validation statement in the check. This means that if three different output variables are created in the applicability and there are two statements in the validation, each of these three created statements is checked twice. If a check operator should only be applied to a certain number of statement blocks, this can be specified by naming the numbering of the statement block at the end of a check statement (more details on this topic can be found in section 6.3.3).

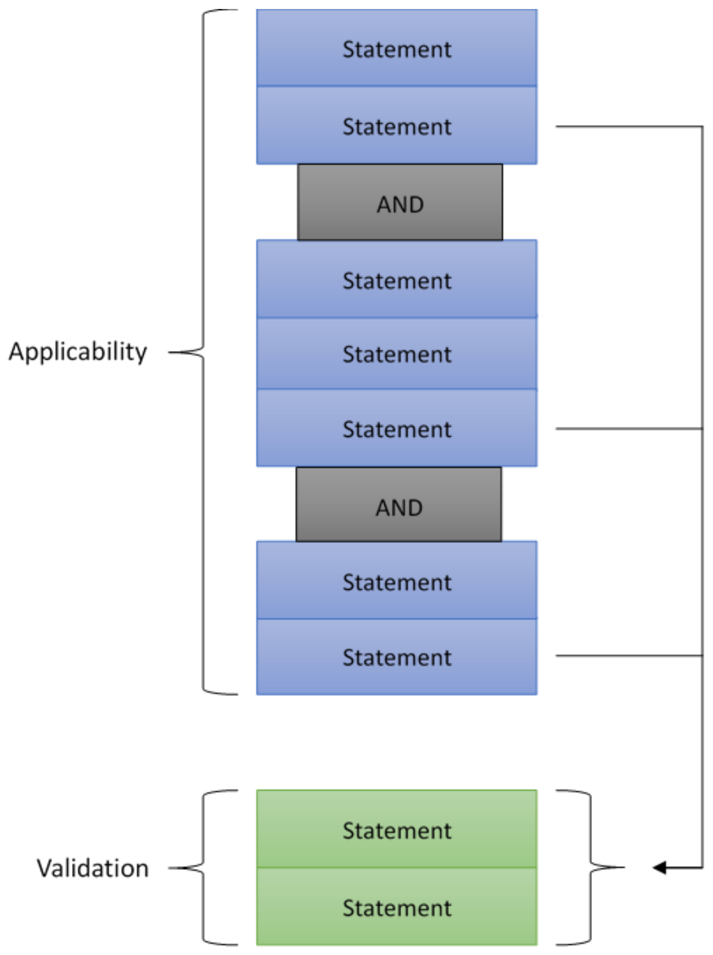


Figure 6-2 Conceptual schema of EIMC-VP

Within one applicability part the use of operators with more than one input variables can mean that several subgroups have to be created. In this case a stringent passing of variables to the next statement is not sufficient. By means of separating parts of the query, blocks can be constructed, whose respective last output variable serves as one of several input variables for the corresponding operator that needs more than one input variable. If a sequence of statements within a statement block should contain several relations, it is further possible that in a statement with several input variables, it should be selected which attribute of a relation should be used for the statement with several input variables. In this case one or more blocks attached to the statement with multiple input variables can be used to specify which attribute of a relation is to be used for the new relation to perform the task. All other attributes of the relation are still included in the new relation. Thus, it can be ensured that these attributes can be used in possible further operations (more details on this topic can be found in section 6.3.3).

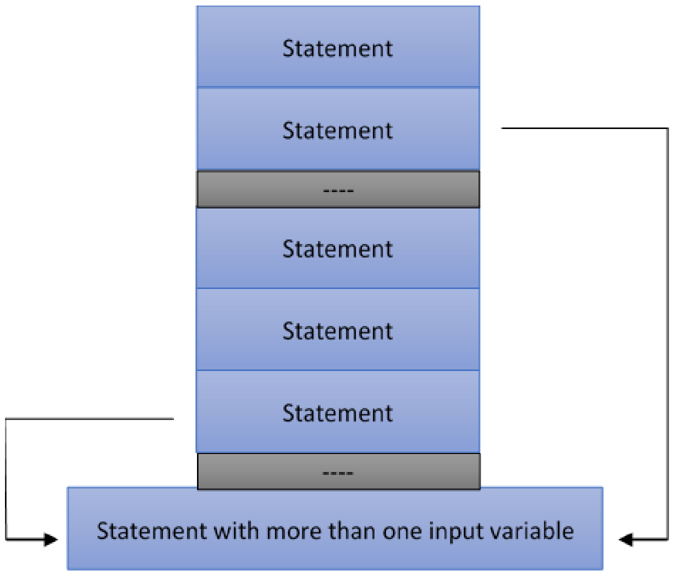


Figure 6-3 Conceptually statement creation of one statement block with more than one input variable

6.3 Concept realisation

In this section the concept realisation of EIM-VP with Blockly is dealt with. First the surface of the user interface of EIMC-VP is explained briefly and then individual blocks are presented. These blocks represent cumulative parts of the textual code of EIMC. Then limitations caused by the facilitate through omitting variables and the resulting workflows to bypass them are shown. This section concludes with two use cases which clarify the creation of queries in EIMC-VP. The representations shown in this section correspond to the current user interface of EIMC-VP. However, due to pending implementations, not all concepts are executable with the prototype at the current state. The limitations due to implementations are discussed in Chapter 7.

6.3.1 Workspace, toolbox, and structuring blocks

When the application is started the user sees both the Toolbox-Area and the Workspace-Area. The workspace is used to combine different blocks to a complete query. In the toolbox the user can choose from different category tabs. These categories contain blocks specific to certain features. With a simple drag and drop procedure blocks can be dragged from the toolbox into the workspace. Four fields in the application's baseboard allow further functions. The button 'Code' shows the created intermediate code for the query, which is currently in the workspace area. The button 'Execute' enables the execution of the query, whereby the 'Status' field can be used to check whether the query has already been executed in the backend. If the execution is completed, the user can download the generated check by clicking on the link Report File.

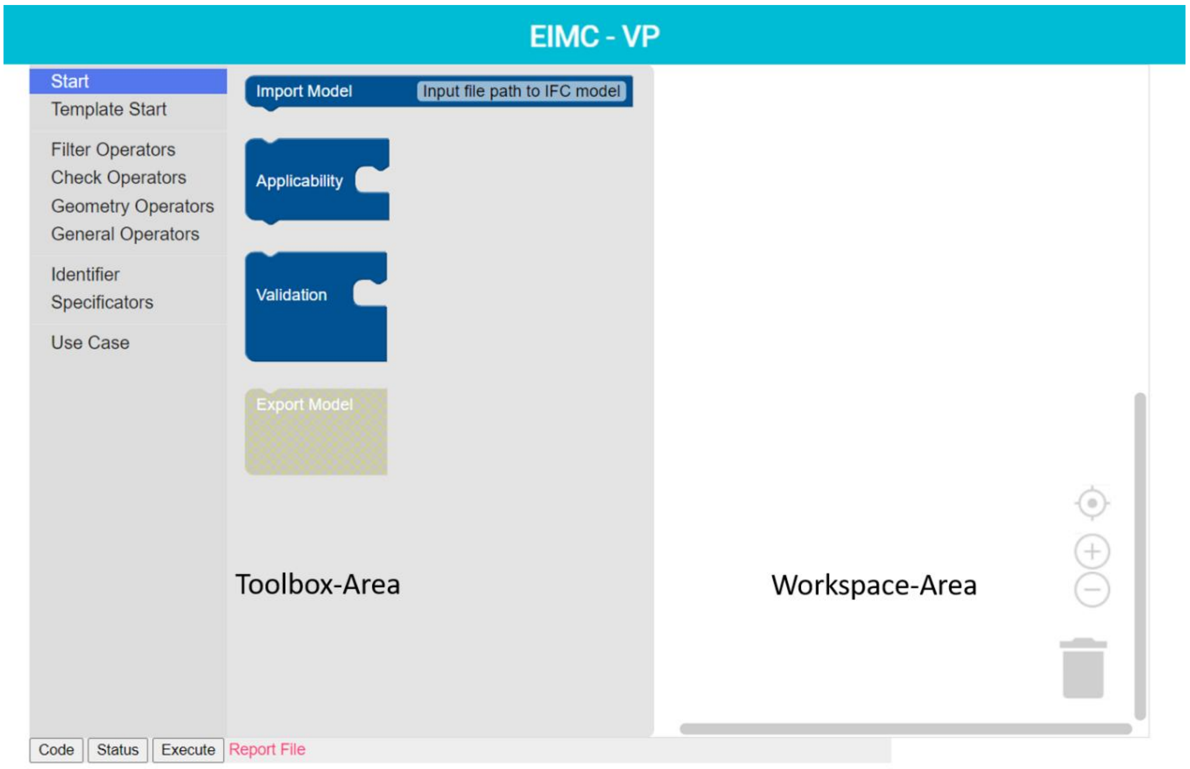


Figure 6-4 Workspace and Toolbox area of the user interface. Start tab opened, showing all structure block

The first category in the toolbox contains all blocks that define the structural framework of a query. That query always starts with the ImportModel block, which imports the entities of the IFC model. Several applicability parts as well as one validation part can then be used to embed a sequence of statements in them. The constructed statements within the structure blocks define how the imported IFC model should be analysed. Structure blocks give users a starting point for the creation of the request and thus ease the use of the visual programming. In order to further simplify the creation and not to create syntactically incorrect queries, there are certain rules for connecting blocks. Blocks have restrictions on how they can be combined with other blocks to ensure that only blocks that result in a syntactically correct query are strung together. In the case of the structuring elements, for example, several Applicability parts can be assembled by combining the blocks one below each other, but just one ImportModel and one Validation part can be used in one query. If users want to carry out a validation, the Validation part is simply attached to the last Applicability part. A further rule enables that in Applicability and Validation parts only operators can be used that are intended for the respective structure block. For example, filter operators can only be used in an

Applicability part and check operators only in a Validation part. The second template of the toolbox contains a predefined template of structure blocks, this template can be used as a starting point for building a sequence.

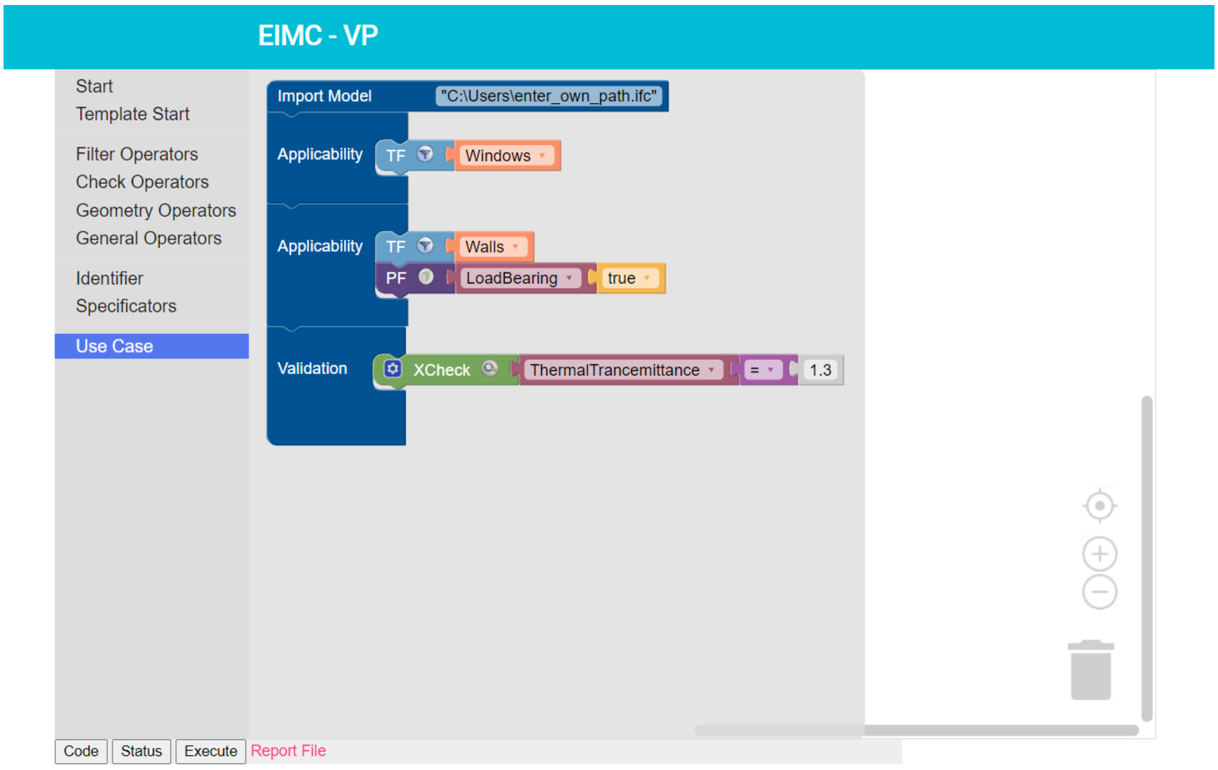


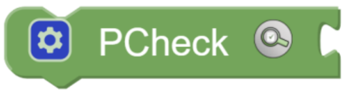
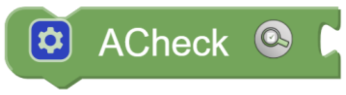
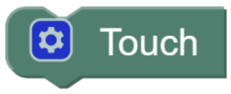
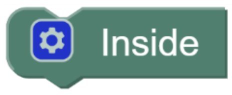
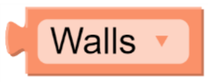
Figure 6-5 Use Case tab opened containing a template of a completed query

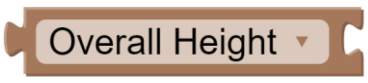
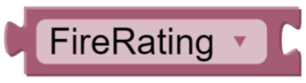
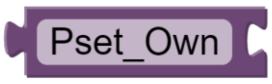
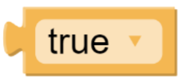
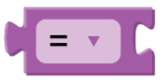
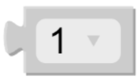
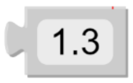
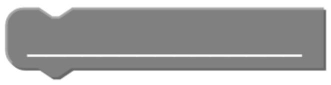
The last tab Use Case contains an already compiled query. This can be easily transferred from the toolbox area to the workspace via drag and drop for further processing. The use case shows users directly which options are available to him and how blocks can be linked together (at the current version only one simple use case is included in the tab, further extended use cases are in progress). The example already contains two applicability parts to indicate that several applicability parts can be lined up one below the other in order to expand different arguments that will be validated separately in the check.

6.3.2 Blocks to create statements in EIMC-VP

The structure blocks (Import Model, Applicability, Export Model or Validation) already presented in the previous section provide the basic framework for the query. Within these blocks, statements are built up to define how a model should be analysed. Each statement starts with an operator. The argument of the operator is constructed horizontally with Identifier and Logical Specification blocks. Each block in EIMC-VP represents different functionalities that are noted in the following Table 5. Operator blocks are additionally highlighted in which part they can be used, whereas Identifier blocks and Logical Specification blocks can be used in both the Applicability parts and the Validation part. A selection of the most important blocks is shown in Table 5, alongside the corresponding name in EIMC and a brief explanatory description.

Block in EIMC-VP	Name	Description
	Operators	
	TypeFilter	Filters the inputted entities from previous statement to specific types. The types are declared in the horizontally next block. Utilisation in Applicability parts.
	PropertyFilter	Filters the inputted entities from previous statement to specific those with specific properties or property sets. The properties are declared in the horizontally connected blocks. Utilisation in Applicability parts.
	AttributeFilter	Filters the inputted entities from previous statement to specific those with specific properties or property sets. The properties are declared in the horizontally connected blocks. Utilisation in Applicability parts.
	XCheck	Checks the last variable of each Applicability part. The gearwheel button enables specification for certain applicability parts. Both properties and attributes can be checked. The check is determined by the horizontally connected blocks. Utilisation in the Validation part.

	PropertyCheck	Checks the last variable of each Applicability part. The gearwheel button enables specification for certain applicability parts. The check is specified on properties and property sets. The check is determined by the horizontally connected blocks. Utilisation in the Validation part.
	AttributeCheck	Checks the last variable of each Applicability part. The gearwheel button enables specification for certain applicability parts. The check is specified on attributes. The check is determined by the horizontally connected blocks. Utilisation in the Validation part.
	TouchOperator	Combines touching entities from previous statements into a relation. Input variables are determined using the Separator (explained in Table 5 paragraph Specificators). Input variables consist of the output variable of the statement directly above a TouchOperator and the output variable directly above a Separator. Utilisation in Applicability parts.
	InsideOperator	Combines entities from previous statements which are topologically inside another entity to a relation. Input variables are determined using the Separator (explained in Table 5 paragraph Specificators). Input variables consist of the output variable of the statement directly above a TouchOperator and the output variable directly above a Separator. Utilisation in Applicability parts.
Identifiers		
	TypeID	Selection of different IFC entities possible in a drop-down menu. Last selection allows to enter own string into the field. Usually attached horizontally to the

		TypeFilter to specify a type. If own text is entered, it must be ensured that the name is correct (e.g. IfcDoors).
	AttributeID	Selection of different IFC attributes possible in a drop-down menu. Last selection allows to enter own string into the field. If own text is entered, it must be ensured that the name is correct (e.g. IfcOwnerHistory).
	PropertyID	Selection of different IFC properties possible in a drop-down menu. Last selection allows to enter own string into the field. If own text is entered, it must be ensured that the name is correct.
	PsetID	Selection of different IFC property sets possible in a drop-down menu. Last selection allows to enter own string into the field. If own text is entered, it must be ensured that the name is correct.
Specificators		
	BooleanSpec	Drop down menu with the choice of true or false. Furthermore, it is possible to select the value exists. Concludes a statement.
	ComparisonSpec	Drop down menu with a selection of comparison operators.
	IntegerSpec	Drop down menu with a selection of integers. Concludes a statement.
	NumberSpec	Text field enables to enter any number that is
	Separator	Allows to specify input variables for geometrical operators. The output variable of the statement above the Separator is

		used as input variable for the next geometrical operator. The statement below a Separator begins, like a statement at the beginning of the Applicability part, with all entities of the IFC model as input variables.

Table 3 Enumeration of the most important blocks in EIMC-VP with names and explanation of functions

The following Figure 6-6 shows the use of different blocks within two statements. The first row illustrates the filtering of all entities from the previous statement to only entities of IfcWall. In a second statement these elements are further filtered. With the PropertyFilter the entities are restricted to those that have attached a property named IsExternal and the value 'TRUE'.

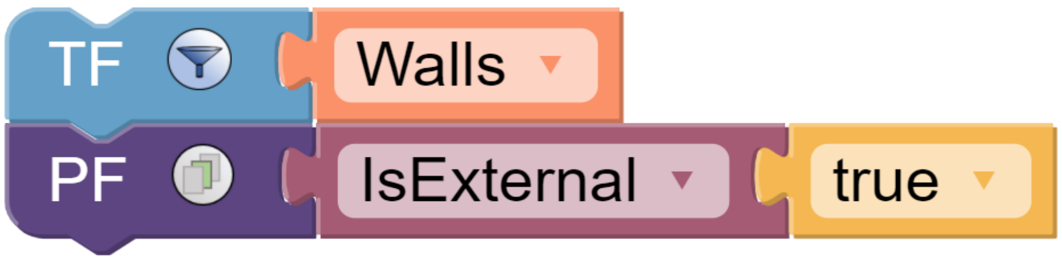


Figure 6-6 Use case of statements below each other in EIMC-VP. The two statements reduce all entities of the previous statement to entities which are external walls

6.3.3 Enhanced workflows

By omitting the variables in Blockly there are some limitations of the textual representation of EIMC that have certain solutions. These solutions usually only occur with longer and more complex queries. In order to be able to display the widest possible range of functionality of the textual EIMC version, these additional functionalities were implemented. Due to the fact that they are rarely used, the additional functions are initially not visible on the blocks. They can be added by a simple procedure. Certain blocks have a small bluish button with a gearwheel symbol. This is used to change the shape of the block in order to extend its functionality when needed. The following Figure illustrates the process of a functionality expansion on a block.

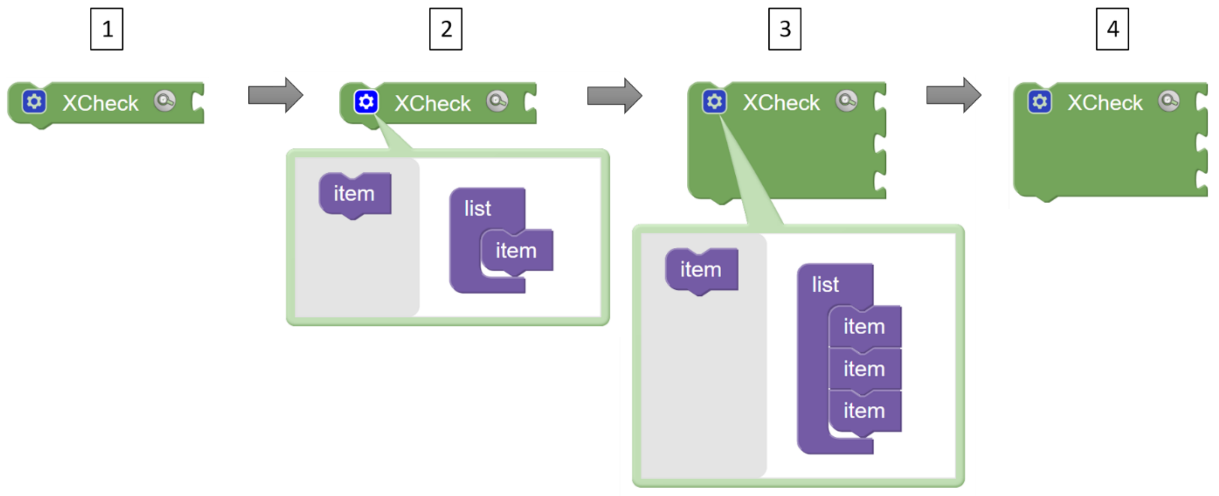


Figure 6-7 Functional expansion of a block demonstrated using XCheck as an example

Figure 6-7 shows in (1) a default shaped XCheck block. Clicking on the gearwheel opens a dialogue field (2). Within this dialogue field blocks can be dragged and dropped, similar to the normal workspace. If, for example, two more item blocks are placed in the list block (3), the form of the XCheck changes immediately. Two new output connectors have been created, to which further blocks can be attached. By clicking on the gear wheel a second time, the dialog field disappears and the block can be used in its new form. Possible applications of the gearwheel button are shown below.

In EIMC-VP, statements that belong together are executed directly one below the other. Output variables of a previous statement serve as input variables of the next statement. By using several applicability parts among each other, several direct passes can be built, where each Applicability part starts again with all entities from the ImportModel operator. Every last output variable of these Applicability part is used as input for all check operators. If three Applicability parts and two check operators are used in the query, a total of six checks are performed. For the output variable of the first, the second and the third Applicability part both check operators are executed as default. If it is wanted to use only one statement block for one of the two check operators, an additionally mark has to be done. Integer blocks can be attached to a statement in the validation part. This has the effect that only the Applicability part is used for the specific check. It is also possible to set several of these blocks one after the other to make a multiple selection.

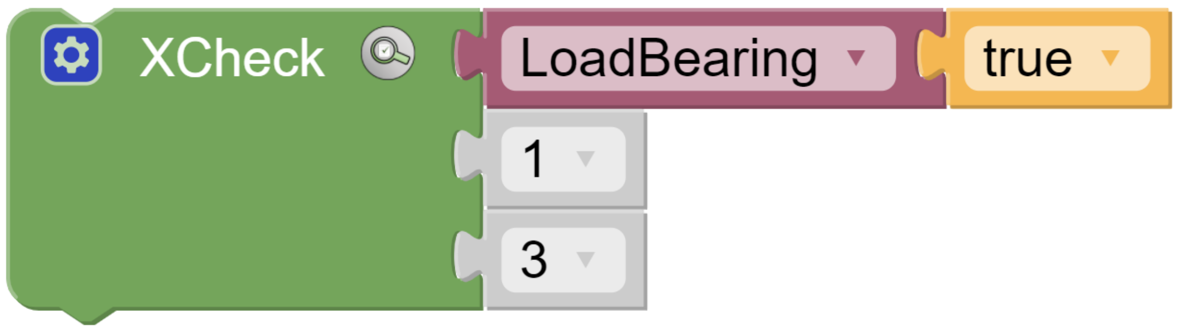


Figure 6-8 Check statement restricted to the first and third Applicability parts

Figure 6-8 shows a check statement with two attached blocks. The XCheck operator checks whether variables (which are built up in the applicability part) serve as input for the operator have a property `LoadBearing` and whether this property has the assigned value true. With the attachment of two additional blocks it is stated that only the output variable of the first and third Applicability part should be used for this explicit check. The output variable of the second Applicability part and possibly further Applicability parts are not initiated as input variable of the check block.

In addition, the connectors created with the dialog field can be used even further. If an applicability part is selected that has a relation as the last output variable, attributes of the relation can be selected using integer nodes. This way only these specific columns of the relation are validated in the check. For example, an Applicability part is selected, which contains a relation of touching walls and floors. The attribute can be specified via index with further, blue-coloured integer blocks in the same line as the grey integer blocks. In the default setting both attributes (walls and floors) would be checked. If the number 2 is appended to the integer node, only the second column is checked. In the example just the instances of the class floors would be checked. The instances of the first attribute, the walls, would not be checked.

The third use of the gear wheel field refers to geometrical operators. For geometrical operators, a manual extension of the block is possible analogous to the process shown in Figure 6-7. The extension is needed if a relation should serve as input variable of a geometrical operator. If a relation is used as input variable, the user can access the indices of the attributes of the relation. If, for example, a pentavalent relation is used as input variable for a TouchOperator, the integer blocks can be used to select which attribute of the relation is to be selected in order to be checked for touches with another input variable. All other attributes of the relation are also included in the new created relation. If an attribute of the pentavalent relation is selected and combined with another set variable, the output variable of the operator is a hexavalent relation. Both the set variable and the entities of the selected attribute touch each other. The other four attributes are included in the new relation in order to be usable in following operations.

6.4 Use cases

In this section two use cases are shown. The first is a simple and straight forward use case. In the second one, an extended use case is shown. They serve to illustrate conceptually the creation of a query and clarify the functionalities within it.

6.4.1 Simple use case

The first use case returns a query that checks all entities of IfcWindow from the IFC model. Each Instance of IfcWindow is checked if the assigned value of the property ThermalTrancemittance is equal to 0.8.

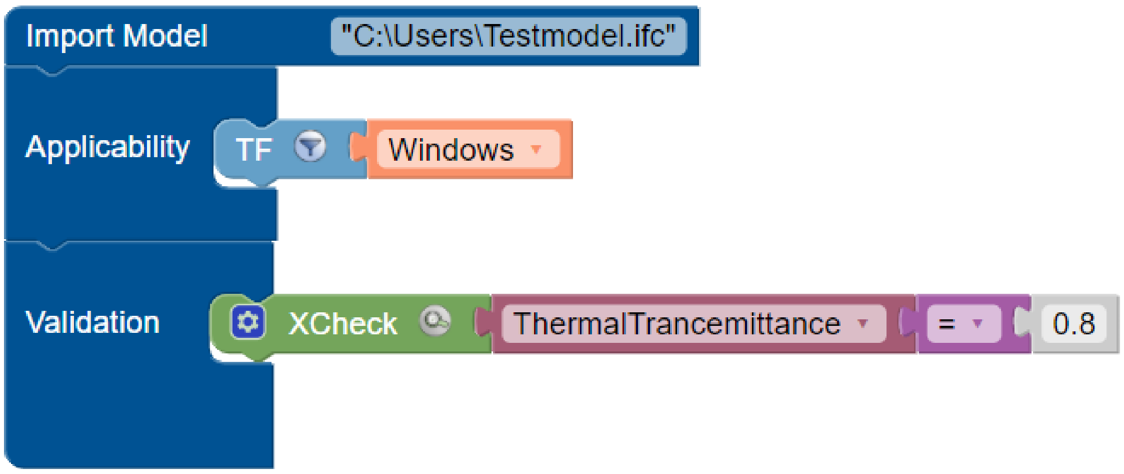


Figure 6-9 Simple use case of a query in EIMC-VP

The query in Figure 6-9 starts with the import of an IFC model. To do this, the path of the storage location is copied into the text line of the ImportModel block. All entities of the IFC model are then passed to the TypeFilter as input variables in the Applicability part. The operator filters the transferred entities to those of IfcWindow. Thereupon, the selected entities are checked in the Validation part. The XCheck examines the properties of the passed entities. It checks if the assigned value of ThermalTrancemittance is 0.8. Once the query is executed and the process completed, the result for each checked entity is written to a JSON file. That report is available as a download link in the user interface of EIMC-VP. For explanation, it should be added that the XCheck examines both attributes and properties. If one only wants to check properties, due to a possibility of the same name with attributes, the PropertyCheck should be used instead of the XCheck.

6.4.2 Expanded use case

The expanded query selects entities in two applicability parts. The selected entities are checked in the validation part to see if the properties FireRating and ThermalTrancemittance exist.

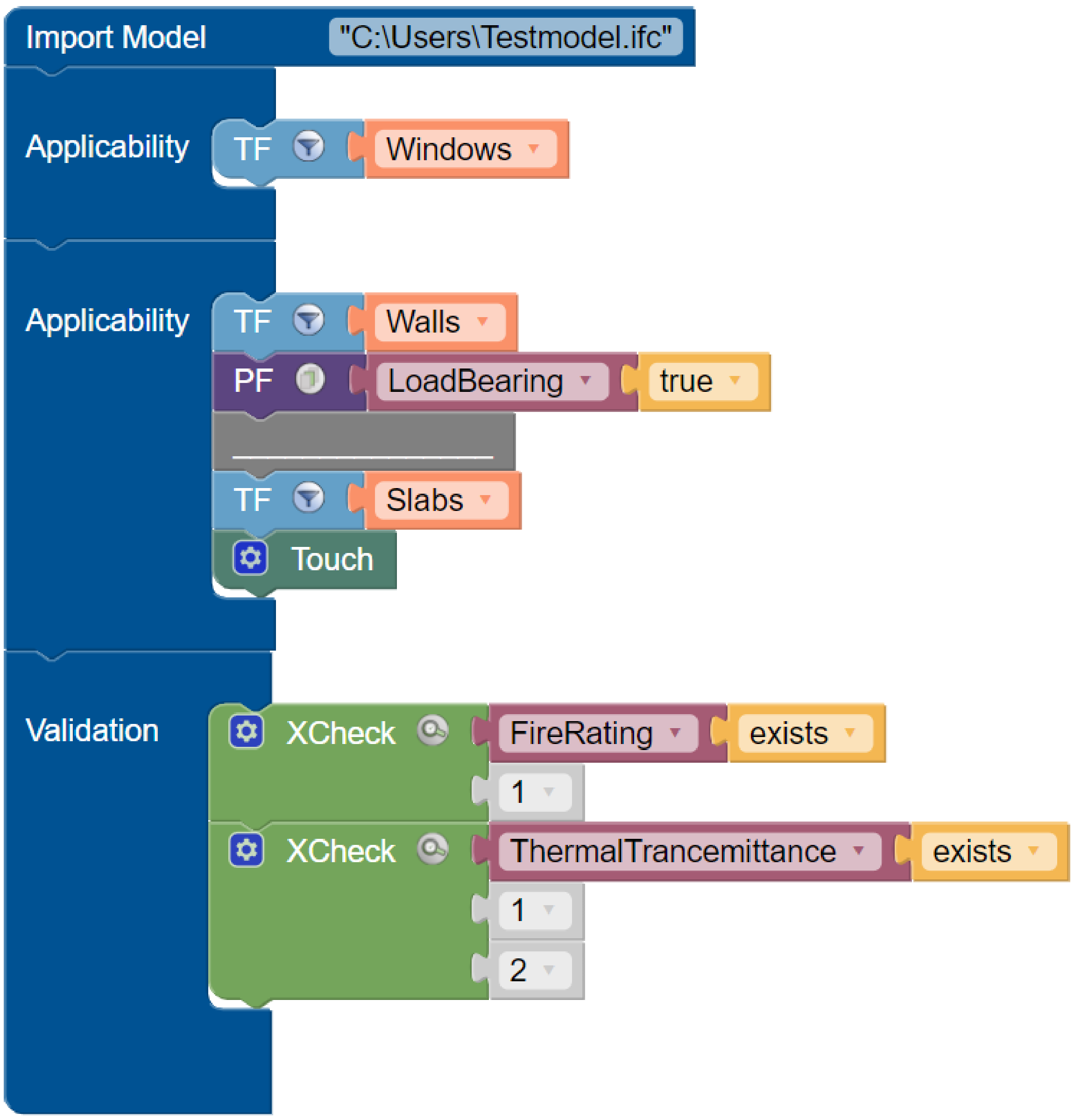


Figure 6-10 Extended use case of a query in EIMC-VP

Figure 6-10 already shows a much more detailed query. All entities of an IFC model are imported by using the ImportModel block. These entities are passed to both Applicability parts and act as input variables of each first statement. The first Applicability part filters all entities to all instances of IfcWindow using the TypeFilter combined with TypeID identifier block. The second Applicability part starts again with all entities of the IFC model as input variable for the first statement. The first operator restricts these entities to all instances of IfcWall using the TypeID identifier block. The following PropertyFilter in the next line specifies the selection of the instances further. The output variable of the PropertyFilter contains all instances of IfcWall that have a property LoadBearing with the corresponding value true. The following separator block indicates that a geometrical operator appears in the sequence. The output variable of the statement directly above the separator block is buffered. The following statement starts again with all entities of the IFC model as input variable. From these, all instances of IfcSlab are selected, the output variable is buffered as well. The TouchOperator now uses the two buffered variables as input variables and creates a relation of entities touching each other. The created relation lists pairs of touching walls and slabs. The generated tuples are stored in the output variable of the second Applicability part. The following Validation part contains two XCheck statements. These two are processed independently of each other. The first check has one integer node attached. Therefore, the check is just applied to the Applicability part declared with the integer block. The first XCheck is therefore only applied to the first Applicability part. The first Applicability part provides all instances of the class IfcWindow. It is checked whether the property FireRating exists by the instances of IfcWindow. Then the second XCheck follows. It performs two individual checks, since two attached integer nodes declare that both the first and the second Applicability part are checked (the same result would be achieved if no integer nodes were attached at all. For a better explanation of the integer nodes, the more detailed variant was chosen in this example). The first of the two performed checks refers to applicability part one and thus checks all windows whether the property ThermalTrancemittance exists. The second performance of the second check refers to the variable obtained from Applicability part two. This provided variable in Applicability part two is now a relation and not a set variable. In the case of the given relation, both the selected load bearing walls and the selected slabs are checked. So, the load bearing walls and the touching slabs are checked for the existence of the ThermalTrancemittance property.

7 Implementation

This work aimed at implementing a prototype. In this Chapter implementations to date will be summarised and limitations of the prototype are outlined, whereby the first section will give an overview of the execution of the application EIMC-VP. Subsequently the implementations in Blockly will be presented. This is followed by an explanation of the implementation of the QL4BIM-Extended-Code-Generator. The Chapter will conclude with a look at the implementation of the validation operators in QL4BIM.

7.1 Execution of EIMC-VP

In the frontend users assemble blocks in EIMC-VP to a query. To execute this, the data entered by the user is first sent to the backend as intermediate code. The intermediate code is a simple JSON format. In the backend the intermediate code is converted into QL4BIM extended code by an implemented code generator. Then the code is forwarded to the QL4BIM server, where the validation is executed on the IFC model. A check report is sent back to the frontend with a record of the validation. Appendix B shows a report generated in this way. Figure 7-1 visualises the general overview.

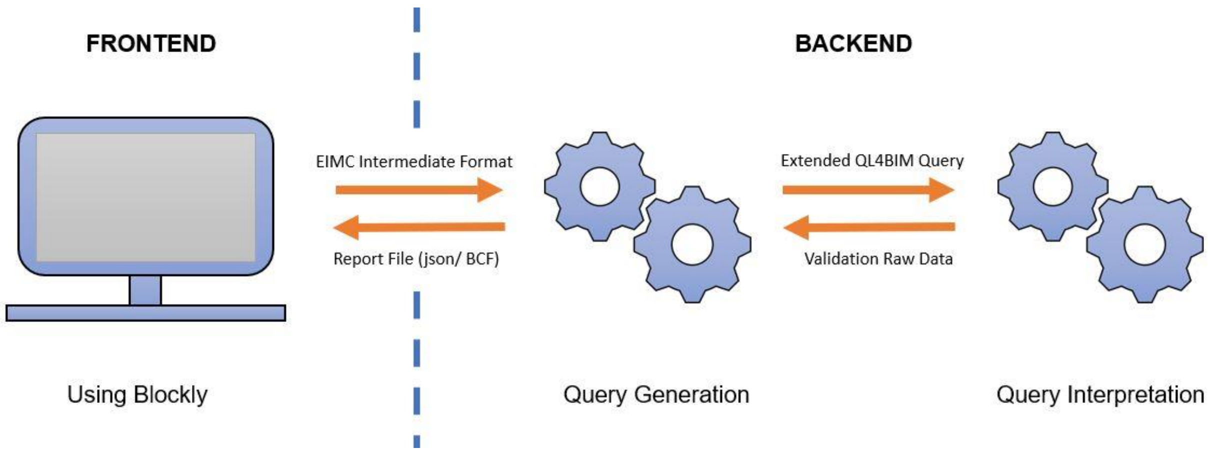


Figure 7-1 Simplified execution of a query in EIMC-VP

Figure 7-2 explains the execution in a more detailed view. After the user has called the webpage, the command of the frontend GET baseurl is sent to the backend. The users can now use the website. They create a query with the available blocks and upload the IFC file whereon they want to execute the query. When the result is executed, the code, which is visible to the user in blocks, is sent to the backend and from there on to the code generator. The code in Blockly is an intermediate code that is represented in JSON. The structure of the intermediate code follows a JSON file. This format has been chosen for its clarity. The JSON structure is converted into a pure QL4BIM extended query in the QL4BIM-Extended-Code-Generator. The converted QL4BIM extended code can then be sent to the QL4BIM extended server, where the code is executed and applied to the IFC file. With continuous requests from the backend to the QL4BIM extended server it is possible to determine when the execution is finished. A report containing detailed information about the check is returned.

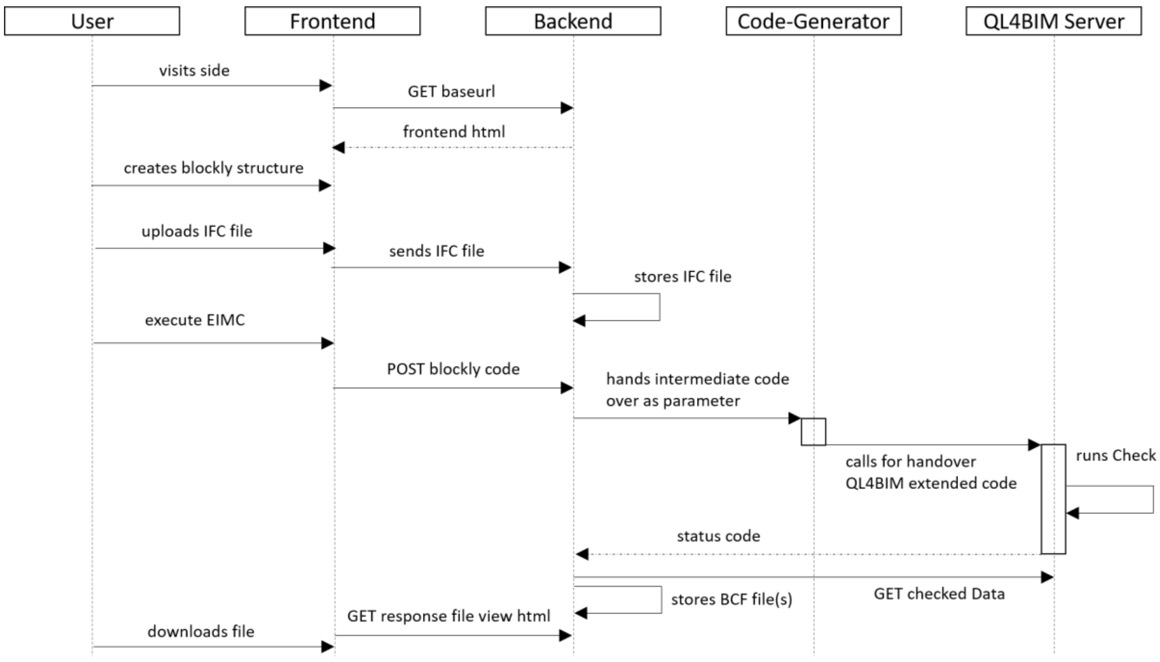


Figure 7-2 More detailed view of the execution process of EIMC-VP

In further work the check report could be converted to a BCF. A BCF file is an open file format that is widely used in the industry. It is used to document issue tracking within a digital building model. The current version of the prototype generates a report in JSON file format, which includes a review of the performed check. Further work is needed to convert the JSON file into a BCF file. When the validation is executed, the intermediate code is posted. In the post request, the request body contains the JSON formatted data. Subsequently, the file is saved in the backend. The frontend receives the information of the successful saving by a continuous request via GET response. The users can finally download the file via a download link in the frontend.

7.2 Implementations in Blockly

Blockly was used for creating a visual programming language. Two of the main reasons to choose especially Blockly were its customizable and extensible implementation and the already implemented user interface. For the realisation of the prototype the official Blockly Guideline (Blockly, 2020a), as well as the GitHub repository of Blockly (Google, 2020), which contains a lot of useful examples to start with and extend, were helpful and served as a basis for the implementation. Due to the specificity of the individual parts in EIMC-VP, all blocks have been newly implemented. For these custom blocks, code generators were implemented in Blockly which generate an intermediate code. This intermediate code follows the data structure of a JSON file. Since the code must ultimately exist in QL4BIM format to be executed by the QL4BIM extended server, an implemented code generator in the backend converts the intermediate code to QL4BIM extended code. First, the implementation of own blocks in the Blockly environment using JavaScript is explained by a simple example, after that more complex implementations like mutators are briefly introduced.

7.2.1 Defining custom blocks in Blockly

In order to explain how JavaScript can be used to create your own blocks in Blockly (Blockly, 2020b), the procedure is shown using the TypFilter operator block in EIMC-VP.

```

Blockly.Blocks['type_filter'] = {
    init: function() {
      this.appendValueInput("NAME")
          .setCheck(null)
          .appendField("TF")
          .appendField(new Blockly.FieldImage
            ("https://abload.de/img/typefilterpngxlkp2.png",
             15, 15, { alt: "bild kann nicht geladen werden", flipRtl: "FALSE" }));
      this.setPreviousStatement(true, "Query");
      this.setNextStatement(true, "Query");
      this.setColour("#64A0C8");
    this.setTooltip("TypeFilter: Operator filters Ifc types");
    this.setHelpUrl("");
    }
};

```

Listing 7-1 Implementation of a custom defined block with JavaScript in Blockly

The first line 391 initiates the reference name for the created block. In the example above this is declared with ‘property_filter’. With the *init* function in line 392 the form of the block is defined. By means of the keyword this, various adjustments can be made. So, in line 393 - 396 an input possibility for a next puzzle piece in one line is created. The visible name on the field is created with the *appendField* function in line 395 and set to ‘TF’ in the example which is an abbreviation for TypeFilter. Additionally, in the following line 396 a picture is inserted into the block. This image serves for easy reuse of the block. Only for the operators that execute a basic start function in QL4BIM and respectively in EIMC, the images have been integrated as icons so that it does not lead to confusion. If you do not want to connect blocks next to each other in one line, you can also stack them on top of each other. The next two functions initiate this. Line 397 *setPreviousStatement* creates a link possibility at the top of the block. This linking possibility is specified by the argument to ‘Query’. Similarly, the function *setNextStatement* in line 398 creates a linking possibility at the lower end of the block and again a specification is set to ‘Query’. The specification allows only certain blocks to be linked together. For example, another TypeFilter operator can be appended to the TypeFilter operator. Likewise, a PropertyFilter or AttributeFilter can be appended (both have as well ‘Query’ as the argument of *setPreviousStatement and setNextStatement*). For validation operators, however, a new variable called ‘Check’ has been introduced instead of the variable ‘Query’. This means that neither a TypeFilter operator can be connected above nor below a Check operator. In line 401 the colour of the block is determined. The hexadecimal colours allow to choose from a wide set of colours. In the last line help instructions are defined. Line 400 defines a tooltip which creates a string that is displayed when the mouse is moved over the block. It serves as an aid for the user. Figure 7-3 shows the code as a block when the code is executed and opened in a webpage.

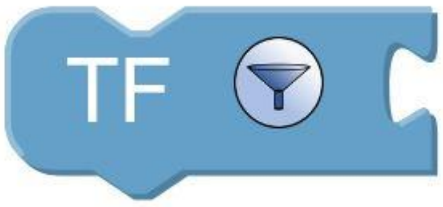
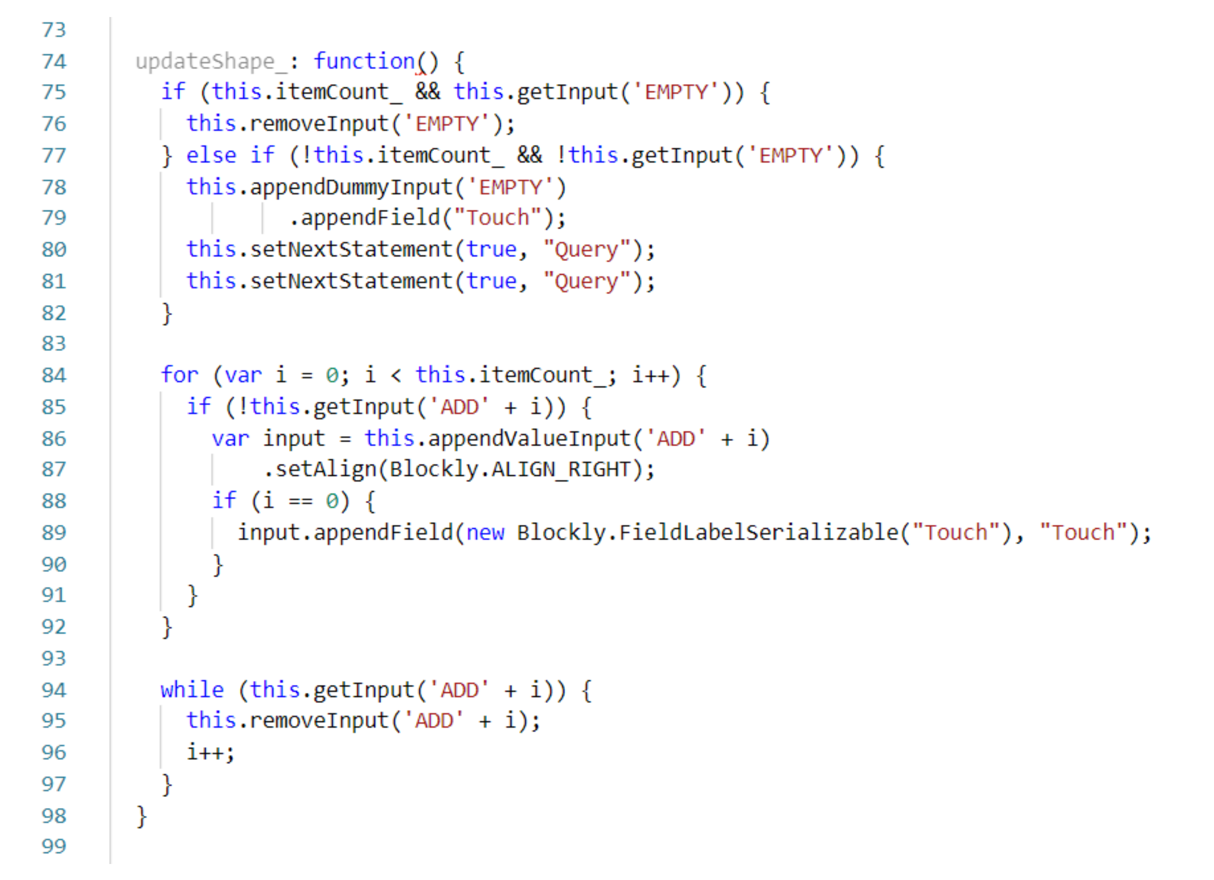


Figure 7-3 Block generated by the JavaScript file from Listing 7-1

In addition to a large number of block types, similar to the type_filter block, two extended block types have been implemented. These extended blocks have a dialog option in Blockly, which enables the user to extend the shape of the blocks. The two block types were implemented to meet the conceptual requirements of section 6.3.3. These requirements demand certain additional behaviours of blocks in certain situations. Both geometry operator blocks and check operator blocks need to contain this extension possibility. It is implemented in Blockly by means of mutators. For the implemented mutators, stable codes from Blockly were used as templates and adapted to the special use cases. The original code can be found in the GitHub repository (Google, 2020). The following code excerpt from the implementation of the touch operator block generates the visible change of the block for the user in the EIMC-VP workspace. In lines 75-81 a block without horizontal output is created. The function to set up a block without any horizontal output is *appendDummyInput* in line 78. In lines 84-89 a for-loop is then initiated, which creates a new horizontal output field in the Blockly editor for each selected ‘ADD’ in the dialog window using the function *appendValueInput*. The While loop in lines 94-96 ensures that a selection can be reversed and that the remove is also made in the EIMC-VP workspace.

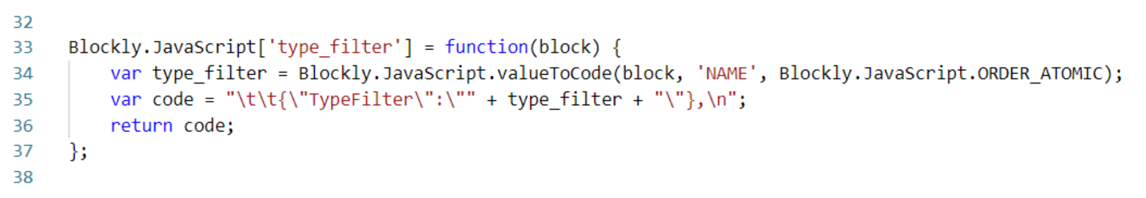


```
73
74    updateShape_: function() {
75      if (this.itemCount_ && this.getInput('EMPTY')) {
76        this.removeInput('EMPTY');
77      } else if (!this.itemCount_ && !this.getInput('EMPTY')) {
78        this.appendDummyInput('EMPTY')
79                .appendField("Touch");
80        this.setNextStatement(true, "Query");
81        this.setNextStatement(true, "Query");
82      }
83
84      for (var i = 0; i < this.itemCount_; i++) {
85        if (!this.getInput('ADD' + i)) {
86          var input = this.appendValueInput('ADD' + i)
87              .setAlign(Blockly.ALIGN_RIGHT);
88          if (i == 0) {
89            input.appendField(new Blockly.FieldLabelSerializable("Touch"), "Touch");
90          }
91        }
92      }
93
94      while (this.getInput('ADD' + i)) {
95        this.removeInput('ADD' + i);
96        i++;
97      }
98    }
99
```

Listing 7-2 UpdateShape function changing the shape of a block in EIMC-VP corresponding to certain ‘ADD’ commands of the user in the user interface

7.2.2 Generate custom code in Blockly

To enable a sequence of blocks created in EIMC-VP to be executed, the blocks must be converted into code. First a custom code generator is created in Blockly for each block type in EIMC-VP, secondly the outputted intermediate code is translated into QL4BIM by an external server. The creation of custom code generators in Blockly is explained with one example. The explanation of creating code generators in Blockly is carried out using the same block as in the section above: ‘type_filter’. After that, a longer code generation is shown by a ready-made query of blocks.



```
32
33  Blockly.JavaScript['type_filter'] = function(block) {
34      var type_filter = Blockly.JavaScript.valueToCode(block, 'NAME', Blockly.JavaScript.ORDER_ATOMIC);
35      var code = "\t\t{\"TypeFilter\":\"" + type_filter + "\"},\n";
36      return code;
37  };
38
```

Listing 7-3 Implementation of a code generator for the type_filter block in Blockly

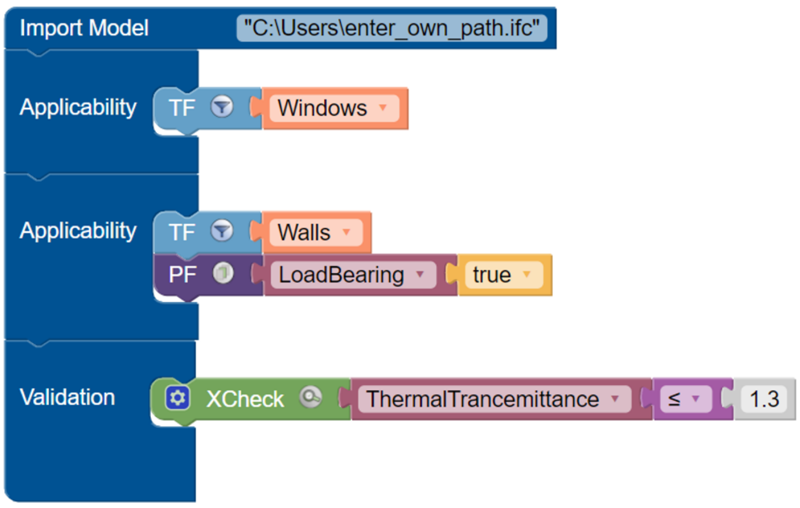
In line 33 the code generator is set up and with the function ‘*return* code’ in line 36 the code is created and returned. Between these two lines, several variables are assigned to different arguments. The first variable in the example is named ‘type_filter’. The variable is defined by the function *valueToCode* which allows to find blocks which have been appended to the corresponding block. The argument 'NAME' describes the name of the corresponding output to be searched. In a second variable with the name *code* the final code is assembled. In the example, first a string is introduced, which is always generated statically in the code when using the block. The string refers to the function of the block followed by a colon; in the example it is 'TypeFilter:' . After that, a variable is inserted into the assembling, separated from the strings by '+' signs. The variable ‘type_filter’, which has been defined before, allows to register adaptively the attached blocks and to interrogate the code generators of these blocks to insert the corresponding code. The code is terminated with a semicolon and a line break. If no further blocks are attached to the TypeFilter block, it will output the code:

```
1 {“TypeFilter”: },
```

Listing 7-4 Generated intermediate code with code generator in Blockly

If several blocks are nested in a query, the above described function *valueToCode* searches all attached blocks and their code generators for the corresponding string output. The appended blocks refer to blocks in one line. If blocks are arranged one below each other, within a statement input block, the function *statementToCode* can analyse the nested blocks and generate the output adaptively to a certain sequence by means of variables in the code (Blockly 2020c). Listing 7-5 shows a sequence of blocks arranged in EIMC-VP and the corresponding generated intermediate code.

A



B

```
{
  "ImportModel": "C:\\Users\\enter_own_path.ifc",

 "Applicabilities": [
    {
      "Applicability": [
        {
          "TypeFilter": "IfcWindow"
        }
      ]
    },
    {
      "Applicability": [
        {
          "TypeFilter": "IfcWall"
        },
        {
          "PropertyFilter": "LoadBearing true"
        }
      ]
    }
  ],
  "Validation": [
    {
      "XCheck": "ThermalTrancemittance <= 1.3"
    }
  ]
}
```

Listing 7-5 Comparison of a query in EIMC-VP (A) and the generated intermediate code (B)

7.2.3 Further implementations in Blockly in future work

The EIMC-VP is currently a prototype. Two features of the frontend are particularly worth mentioning for future work on the prototype.

The first essential function is the appliance of intelligence in the application. If a user creates a sequence of statements, dependencies should be created in the EIMC-VP workspace. This means that if the user wants to create a query that specifies windows to a specific property then the user should have, in addition to a manual input, the possibility of making a selection from the drop down menu of the PropertyID identifier block that is bounded to IfcWindow related properties in the IFC schema. In the current version a drop-down menu is implemented, which contains exemplary for further properties only a short list of properties from the property set Pset_WallCommon. However, the current block allows also to enter an own string into the field, thus ensuring that all predefined properties of buildingSMART as well as properties you have created yourself can be accessed.

Secondly, the restriction of block nesting can be examined more closely. The current version already has some restrictions on the connection of blocks. For example, no filter operator can be connected to a check operator. Also, no filter operator can be nested in the statement block of the validation and vice versa no check operator can be nested in an applicability statement block. Nevertheless, when creating individual statements in a horizontal row in EIMC-VP, it is possible to arrange blocks in such a way that queries cannot be generated syntactically correct. To provide the greatest possible spectrum when creating queries, the restriction of the individual blocks must be precisely defined. Further investigations, how far different blocks should be restricted in their connectivity, are necessary to ensure the best possible usage. If a syntactically incorrect query is built within the user interface, the code generator informs the frontend that this query cannot be executed. The code generator is described in more detail in the following section.

7.3 QL4BIM-Extended-Code-Generator

The QL4BIM-Extended-Code-Generator converts intermediate code from Blockly into QL4BIM extended code. The implementation of the QL4BIM-Extended-Code-Generator was developed in close collaboration and with help from Daum (Sirtl et al., 2020). Further code regarding the Code-Generator, not shown in this section, has been implemented by Daum (Sirtl et al., 2020). A brief overview of the implementation and its functioning is given below.

7.3.1 Implementation of the QL4BIM-Extended-Code-Generator

In the frontend the end user creates a query in EIMC-VP. When the run command is executed, a message with the created intermediate code is sent to the backend. The transmitted data structure is a textual representation of the in JSON format structured intermediate code. Figure 7-4b shows the JSON formatted intermediate code. In the backend, the textual representation of the code is converted back into an object-orientated representation by using Python's JSON-to-Object conversion. The algorithm of the QL4BIM-Extended-Code-Generator then traverses the tree structure. Certain objects, which are nodes in the tree structure, trigger certain generation of QL4BIM extended statements. Some important ones are briefly explained below. First the algorithm checks whether the transmitted query starts with the ImportModel operator. If this is not the case, an error message is displayed in the frontend. If the ImportModel operator is recognised, the first line of QL4BIM extended code is generated. The algorithm then continues to traverse the tree structure. If the key *Applicabilities* applies, the algorithm knows that separate *Applicability* parts must be treated individually in the following. In each of these *Applicability* parts, one tree node represents one statement of extended QL4BIM. Within each *Applicability* part, the algorithm sets the first variable of the query as the input variable for the argument of the first operator. This ensures that each applicability part begins with all available entities of the IFC model in the first statement. The algorithm creates an automatically generated output variable for the first and every following statement. These variables are stored in a stack. Every time a new variable is generated, it is stored at the top of the stack. For each additional node in the tree structure within the same *Applicability* part, the algorithm uses the top variable of the stack as input variable for the next statement. This sequence is interrupted as soon as the *Applicability* branch ends. Further *Applicability* parts are traversed in the same way. The last output variable of each individual *Applicability* part is stored on a list. These variables of the list are finally checked in the *Validation* part. Through the key *Validation* it is recognised that the following part belongs to the validation part. For the following check type operators, the algorithm uses for each new node the generated relation from the variables of the second stack as input for the argument of each operator. The result is a code in QL4BIM extended format. Listing 7-

6 shows the *nodes_traversal* function which offers the mentioned tree traversal functionality.

```
26
27  def nodes_traversal(d, querybuild):
28
29      def iter1(d, parent, pathstack, key, index, querybuild):
30          process_node(d, parent, pathstack, key, index, querybuild)
31
32          if isinstance(d, dict):
33              for k, v in d.items():
34                  local_pstack = list(pathstack)
35                  local_pstack.append("k_" + str(k))
36                  iter1(v, d, local_pstack, k, None, querybuild)
37                  process_node_post(v, d, local_pstack, k, None, querybuild)
38
39          elif isinstance(d, list):
40              for i, v in enumerate(d):
41                  local_pstack = list(pathstack)
42                  local_pstack.append("i_" + str(i))
43                  iter1(v, parent, local_pstack, None, i, querybuild)
44                  process_node_post(v, parent, local_pstack, None, i, querybuild)
45          else:
46              local_pstack = list(pathstack)
47              local_pstack.append("v_" + str(d))
48              process_value(d, parent, querybuild)
49
50
51      iter1(d, None, [], None, None, querybuild)
52
```

Listing 7-6 Code that traverses the tree structure of the transferred intermediate code

The function *nodes_traversal* takes a parameter which is marked with the letter 'd' in line 27 of Listing 7-6. The parameter 'd' can be typed as dictionary, list, or simple value in the recursive tree traversal. The dictionary is a data structure in which JSON can be mapped. It has certain keys which are stored in parameter 'key'. ImportModel, Applicabilies and Validation are among others such keys. In line 29 the iterating function *iter1* is called within *nodes_traversal*. Here the iteration over the tree structure begins. The function *process_node* executes the command how to handle the current node. The algorithm has got to distinguish between dictionaries, lists and simple values. The first If-Condition (line 32) follows commands with the instance that the node to be examined is a dictionary. In the first execution the node is the whole intermediate code, this is a dictionary and therefore the condition follows that the node is resolved. The function resolves the current node and the node, that is now considered is located in the tree structure one level below (line 35). Then line 36 causes the function *iter1* (line 29) to be called recursively. The function checks again in the first If-Condition if the current node is a dictionary, if so lines 32-37 are executed again. If not, the algorithm checks the next possibility in line 39, which is a list. The list is resolved with indices. Each part of the list is assigned to the variable ‘v’ and a variable ‘i’ for the index is also created. The third possibility (lines 45-48) is that the value just treated is a simple data type. The function *process_value* executes specific code depending on simple value data. The traversing of the tree structure of the intermediate code is additionally linked to *process_node* (line 30). *Process_node* is the function that creates the output as QL4BIM code.

```
80
81      if key == "AttributeCheck":
82          pred_parts = predicate_splitter(d)
83
84          #with index or without
85          if len(pred_parts) == 3:
86              querybuilder.addAttCheckOpt(pred_parts[0], pred_parts[-1])
87          elif len(pred_parts) == 4: #with index
88              querybuilder.addAttCheckOpt(pred_parts[0], pred_parts[-2], pred_parts[-1])
89          querybuilder.increase_validation_index()
90
```

Listing 7-7 Excerpt of the process_node function

Listing 7-7 contains an extract from the function. This excerpt contains the case of the key “*AttributeCheck”* (line 80). The function passes the object associated to the *AttributeCheck* key to the querybuilder, which then creates the actual QL4BIM statement and outputs it as a string. The two If-Conditions in line 85 and 87 check whether the check was specified with indices on a certain applicability part or not.

7.3.2 Further implementations in the Code-Generator in future work

The QL4BIM-Extended-Code-Generator converts the intermediate code generated in Blockly into pure QL4BIM extended code which can be executed in the QL4BIM server. There are some restrictions that are not yet possible to convert. The most important one is the ExportModel block. The focus on the automated validation of information using EIMC-VP has the consequence that the implementation of a pure filter output of a model was considered secondary to this work. Further work on the prototype can enable the output of a filtered IFC model. In addition, the implementation of further general operators of QL4BIM is also being considered in future work. One example is the Projector operator. This is used in QL4BIM to resolve relations. In this way, the attributes of a relation can be treated in the subsequent query independently of the other attributes of a relation.

Apart from these restrictions, the comparative block has not yet been implemented. The current implementation status always overwrites the comparison operator with an is-equal. This is due to the fact that it was considered essential to allow a default mode. The default mode allows to omit a comparison operator. For example, this ensures that comparisons with a Boolean operator such as true or false can be realised without an interposed block between the specification of an attribute or property and the Boolean operator.

Similarly, a valid structure of several Applicability parts is not yet possible. Further investigations of the Code-Generator must be carried out so that several Applicability parts can be used among each other to check different entities. However, it has already been possible to execute sample queries using queries on an IFC model that only had one applicability part.

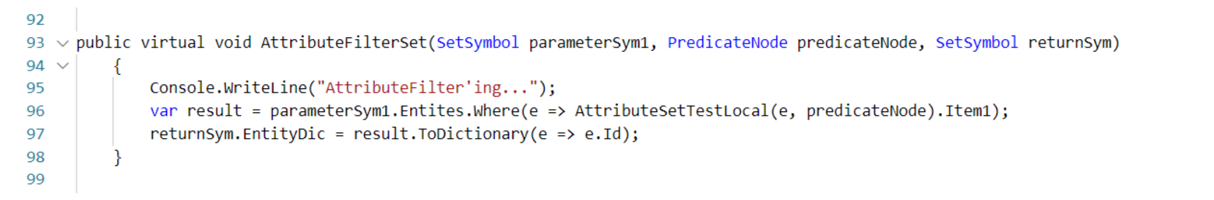
The PropertyCheck cannot yet be converted stably. Also the PropertyFilter is not yet stable, further investigations are necessary to make the conversions work. However, the AttributeCheck and a AttributeFilter can already be executed. The quality of the generated check report must be checked in further work.

7.4 Implementations in QL4BIM

7.4.1 Implementation of the Validation Operators

The development of the validation operators is an important extension of QL4BIM for EIMC. The implementation of the validation operators in QL4BIM was carried out by Daum (Sirtl et al., 2020). The features of the implementation were discussed and agreed upon in close consultation. Short excerpts from the implementation are shown and briefly explained. The AttributeCheck and the PropertyCheck operators have been implemented in C# and embedded in QL4BIM, the XCheck is not yet implemented. The following extracts give an approach to the implementation of the check operators.

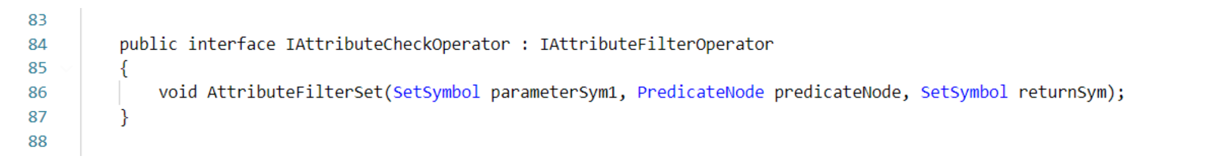
The AttributeCheck is explained exemplarily for all check operators. The implementation of the AttributeCheck operator occurs as one overloading of the AttributeFilter. This enables to reuse the implementation of the low-level examination of attributes and their values. The implementation core of the AttributeFilter is shown in Listing 7-7. In line 93 the *AttributeFilterSet* function is initialized. This contains the connection to the string 'AttributeFilter' (line 95) and a numerical list of the entities (line 97). In line 96 the variable result is initialised for this purpose. This variable receives a set of entities from the function *AttributeSetTestLocal* using the *Where* function. This function checks whether the entity found in the IFC model meets the predicate of the operator.



```
92
93  public virtual void AttributeFilterSet(SetSymbol parameterSym1, PredicateNode predicateNode, SetSymbol returnSym)
94      {
95          Console.WriteLine("AttributeFilter'ing...");
96          var result = parameterSym1.Entites.Where(e => AttributeSetTestLocal(e, predicateNode).Item1);
97          returnSym.EntityDic = result.ToDictionary(e => e.Id);
98      }
99
```

Listing 7-8 Implementation of filtering using AttributeFilter

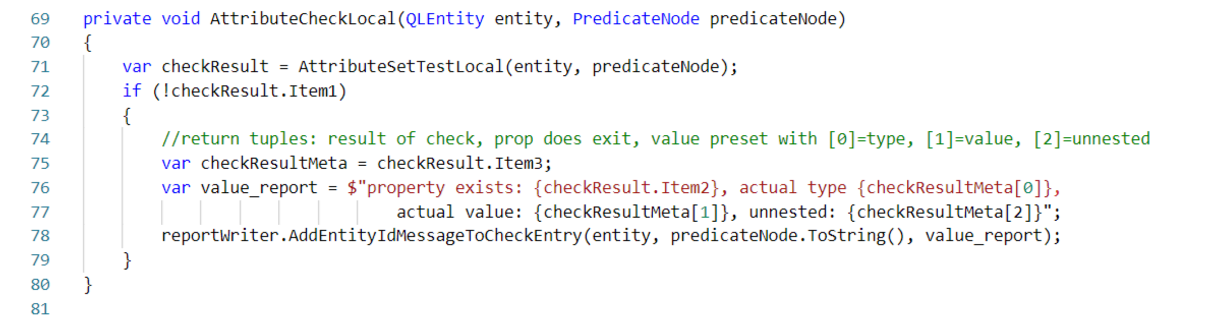
The following Listing 7-8 shows the overwriting of the AttributeFilter with the AttributeCheck, which allows the AttributeCheck to serve as an overloading of the AttributeFilter.



```

    public interface IAttributeCheckOperator : IAttributeFilterOperator
    {
        void AttributeFilterSet(SetSymbol parameterSym1, PredicateNode predicateNode, SetSymbol returnSym);
    }

```

Listing 7-9 Initialisation of the AttributeCheck operator as overloading of the AttributeFilter operator

The report creation is noted in Listing 7-9. It is noted whether the attribute exists ('property exists'), the actual type ('actual type'), the actual value and whether it is ‘unnested’. Which of these values is noted in the report file depends on the argument of the used AttributeFilter.



```
private void AttributeCheckLocal(QLEntity entity, PredicateNode predicateNode)
{
    var checkResult = AttributeSetTestLocal(entity, predicateNode);
    if (!checkResult.Item1)
    {
        //return tuples: result of check, prop does exit, value preset with [0]=type, [1]=value, [2]=unnested
        var checkResultMeta = checkResult.Item3;
        var value_report = $"property exists: {checkResult.Item2}, actual type {checkResultMeta[0]},
                                actual value: {checkResultMeta[1]}, unnested: {checkResultMeta[2]}";
        reportWriter.AddEntityIdMessageToCheckEntry(entity, predicateNode.ToString(), value_report);
    }
}

```

Listing 7-10 Adding the results of the check to the report file of the AttributeCheck

7.4.2 Further Implementations in QL4BIM

Some important elements of EIMC and EIMC-VP are not yet available in QL4BIM. Hence, some queries which are possible to crate in the user interface in EIMC-VP cannot yet be executed.

As mentioned in the previous section, the XCheck operator is not yet implemented, an overloading of the AttributeCheck with the functions of the PropertyCheck operator has to be implemented in QL4BIM. Consequently, a query containing the XCheck cannot yet be executed.

A further one is the ExistCheck function, which has not yet been implemented and needs further development. Therefore the execution of the exist block does not yet work and a query that checks the existence of attributes or properties cannot yet be executed.

7.5 Summarizing the current state

The current prototype still has a number of limitations that significantly restrict its functionality. However, simple queries have already been successfully generated. Queries using TypeFilter, AttributeFilter and AttributeCheck operators and the corresponding statement creation blocks could already successfully execute simple queries and generate a check report. Thus the goal of this work to create a user-friendly MVD generator could be indicated. Further work and testing on the QL4BIM-Extended-Code-Generator and in Ql4BIM are necessary to reflect the current state of the user interface. It is also important to perform further tests to validate the quality of the generated check reports. Especially important is the implementation of the XCheck operator, the ExistCheck opportunity, and to stable the possibility to set several Applicability parts among each other in order to separately check different sequences in one query. Furthermore, the overwriting of the comparison operators must be changed so that the default mode ‘is equal to’ still is supported, but if a different comparison operator is used, it will be generated in the QL4BIM-Extended-Code-Generator.

8 Summary and Outlook

As the amount of information in a building model increases, the importance of automated quality assurance is growing. The MVD method of buildingSMART enables an automated validation of information in a building model. However, due to the complexity of the format it is not possible for domain experts to create an MVD that validates specific data in a certain use case. The presented EIMC concept aims to give an intuitive entry to reproduce functional areas of the MVD Method. The developed application EIMC-VP facilitates the application of the EIMC concept by means of visual programming. The conceptually approach of EIMC-VP enables the user to create an analysis of the digital building model and to automatically validate the information of the model. In summary, the prototype already enables the objective of this work to facilitate automatic validation of information in an IFC instance model. However, it must be clearly stated that further implementations and tests are necessary for the prototype to run stable. At the current state, only simple queries are possible to be executed. First some further implementations must be entered in the prototype, so that important concepts such as the usage of several Applicability parts can be used stably and important features as the ExistCheck are possible. In addition, further tests have got to be carried out in the long run to increase the user-friendliness even more and implement more features of the MVD method. For example, the implementation of an intelligence that allows certain selection focuses based on already created parts of a query. This should make it even easier for the user to select the right blocks and their values in a query is aimed at. It is also intended to integrate further functions of an MVD. This includes the pure filtering of a digital building model.

Bibliography

Abualdenien, J., & Borrmann, A. (2019). A meta-model approach for formal specification and consistent management of multi-LOD building models. *Advanced Engineering Informatics*, 40, 135–153. https://doi.org/10.1016/j.aei.2019.04.003

Abualdenien, J., Pfuhl, S., & Braun, A. (2019). Development of an MVD for checking fire-safety and pedestrian simulation requirements. In *31. Forum Bauinformatik: 11.–13. September 2019 in Berlin. Proceedings*. Berlin: Universitätsverlag der TU Berlin.

Autodesk (2020). *Overview - Get to know Dynamo Studio*. Retrieved: https://www.autodesk.com/products/dynamo-studio/overview?plc=DYNSTD&term=1-YEAR&support=ADVANCED&quantity=17. Accessed: 28.09.2020

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

BIM Supporters B.V. [BIM Secrets]. (02.09.2018). *IFC Schema Basics* [Video]. Youtube. Retrieved: https://www.youtube.com/watch?v=q_7i4l7KSeI&ab_channel=BIMSecrets. Accessed: 28.09.2020.

Blockly (2020a). *Introduction to Blockly*. Retrieved: https://developers.google.com/blockly/guides/overview. Accessed: 28.09.2020.

Blockly (2020b). *Define Blocks*. Retrieved: https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks. Accessed: 28.09.2020.

Blockly (2020c). *Generating Code*. Retrieved: https://developers.google.com/blockly/guides/create-custom-blocks/generating-code. Accessed: 28.09.2020.

Blockly (2020d). *Extensions and Mutators*. Retrieved: https://developers.google.com/blockly/guides/create-custom-blocks/extensions. Accessed: 28.09.2020.

Borrmann, A., König, M., Koch, C., & Beetz, J. (Hrsg.) (2015). *Building Information Modeling Technologische Grundlagen und industrielle Praxis.* Wiesbaden: Springer-Verlag.

buildingSMART (2019a). *What we do*. Retrieved: https://www.buildingsmart.org/about/what-we-do/. Accessed: 28.09.2020.

buildingSMART (2019b). *IFC Formats*. Retrieved: https://technical.buildingsmart.org/standards/ifc/ifc-formats/. Accessed: 28.09.2020.

buildingSMART (2019c). *The curious case of the MVD*. Retrieved: https://blog.buildingsmart.org/blog/the-curious-case-of-the-mvd. Accessed: 28.09.2020.

buildingSMART (2020a). *The importance of Industry Foundation Classes in Building Information Modelling*. Retrieved: https://blog.buildingsmart.org/blog/importantifc. Accessed: 28.09.2020.

buildingSMART (2020b). *IfcPropertySet*. Retrieved: https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2/HTML/schema/ifckernel/lexical/ifcpropertyset.htm. Accessed: 28.09.2020.

buildingSMART (2020c). *Pset_WallCommon*. Retrieved: http://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2/HTML/schema/ifcsharedbldgelements/pset/pset_wallcommon.htm. Accessed: 28.09.2020.

buildingSMART (2020d). *Information Delivery Manual (IDM)*. Retrieved: https://technical.buildingsmart.org/standards/information-delivery-manual/. Accessed: 28.09.2020.

buildingSMART (2020e). *Model View Definition (MVD) - An Introduction*. Retrieved: https://technical.buildingsmart.org/standards/ifc/mvd/. Accessed: 28.09.2020.

buildingSMART (2020f). *MVD Database*. Retrieved: https://technical.buildingsmart.org/standards/ifc/mvd/mvd-database/. Accessed: 28.09.2020.

Chipman, M., Liebich, T., & Weise, M. (2016). *mvdXML: Specification of a standardized format to define and exchange Model View Definitions with Exchange Requirement and Validation Rules.* Retrieved: https://standards.buildingsmart.org/MVD/RELEASE/mvdXML/v1-1/mvdXML_V1-1-Final.pdf, Accessed: 09/09/2020.

Daum, S. (2018). *Konzeption einer raum-zeitlichen Anfragesprache für die Analyse und Prüfung von 4D-Gebäudeinformationsmodellen*. (Doctoral dissertation, Technische Universität München).

Google (2020). google/blockly. Retrieved: https://github.com/google/blockly. Accessed: 28.09.2020.

Microsoft (2015). *Functional programming vs. imperative programming (LINQ to XML)*. Retrieved: https://docs.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming. Accessed: 28.09.2020.

Ritter, F., Preidel, C., Singer, D., & Kaufmann, S. (2015). Visuelle Programmiersprachen im Bauwesen - Stand der Technik und aktuelle Entwicklungen. In *Proceedings of the 27th Forum Bauinformatik, Aachen (Germany)*.

Saake, G., Heuer, A., & Sattler, K. (2018). XML, XQuery und SQL/XML. In Saake, G., Heuer, A., & Sattler, K. (Hrsg) *Datenbanken - Konzepte und Sprachen*. MITP-Verlags GmbH & Co. KG. doi: https://learning.oreilly.com/library/view/datenbanken-konzepte/9783958457782/xhtml/ch30.xhtml.

SimpleBIM (2020a). *Features*. Retrieved: https://simplebim.com/features/. Accessed: 28.09.2020

SimpleBIM (2020b). *Add New Templates*. Retrieved: https://simplebim.com/support/howto-add-new-template.html. Accessed: 28.09.2020.

SimpleBIM (2020c). *mvdXML*. Retrieved: https://simplebim.com/support/addon-mvdxml.html. Accessed: 28.09.2020.

Sirtl, F., Daum, S., & Abualdenien, J. (2018). *EIMC*. Retrieved: https://github.com/FelixSirtl/EIMC. Accessed: 29.09.2020.

Wagenknecht, C. (2016). *Programmierparadigmen: Eine Einführung auf der Grundlage von Racket*. Wiesbaden: Springer-Verlag. doi: 10.1007/978-3-658-14134-9.

World Wide Web Consortium (2016). *Extensible Markup Language (XML)*. Retrieved: https://www.w3.org/XML/. Accessed: 28.09.2020.

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

Appendix A

mvdXML use case

```
<?xml version="1.0" encoding="UTF-8"?>
<mvdXML name="example MVD for mvdXML documentation – sensor signals"
uuid="4afb1a8b-0b61-4ff8-9863-c10690fe06f2"
xmlns="http://buildingsmart-tech.org/mvd/XML/1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://buildingsmart-tech.org/mvd/XML/1.1 ../mvdXML_V1.1.xsd">

<Templates>
    <ConceptTemplate uuid="bafc93b7-d0e2-42d8-84cf-5da20ee1480a" name="Port Assignment"
     applicableSchema="IFC4" applicableEntity="IfcDistributionElement">
     <Definitions>
       <Definition>
         <Body>
            <![CDATA[<p>Distribution ports are defined by <i>IfcDistributionPort</i>
            and attached by the <i>IfcRelNests</i> relationship. Ports can be
            distinguished by the
            <i>IfcDistributionPort</i> attributes <i>Name</i>, <i>PredefinedType</i>,
            and <i>FlowDirection</i>:</p>]]>
          </Body>
        </Definition>
       </Definitions>
       <Rules>
            <AttributeRule AttributeName="IsNestedBy">
                <EntityRules>
                    <EntityRule EntityName="IfcRelNests">
                        <AttributeRules>
                        <AttributeRule AttributeName="RelatedObjects">
                            <EntityRules>
                                <EntityRule EntityName="IfcDistributionPort">
                                    <AttributeRules>
                                        <AttributeRule AttributeName="Name" RuleID="Name"/>
                                        <AttributeRule AttributeName="PredefinedType" RuleID="Type"/>
                                        <AttributeRule AttributeName="FlowDirection" RuleID="Flow"/>
                                    </AttributeRules>
                                </EntityRule>
                            </EntityRules>
                        </AttributeRule>
                        </AttributeRules>
                    </EntityRule>
                </EntityRules>
            </AttributeRule>
       </Rules>
```

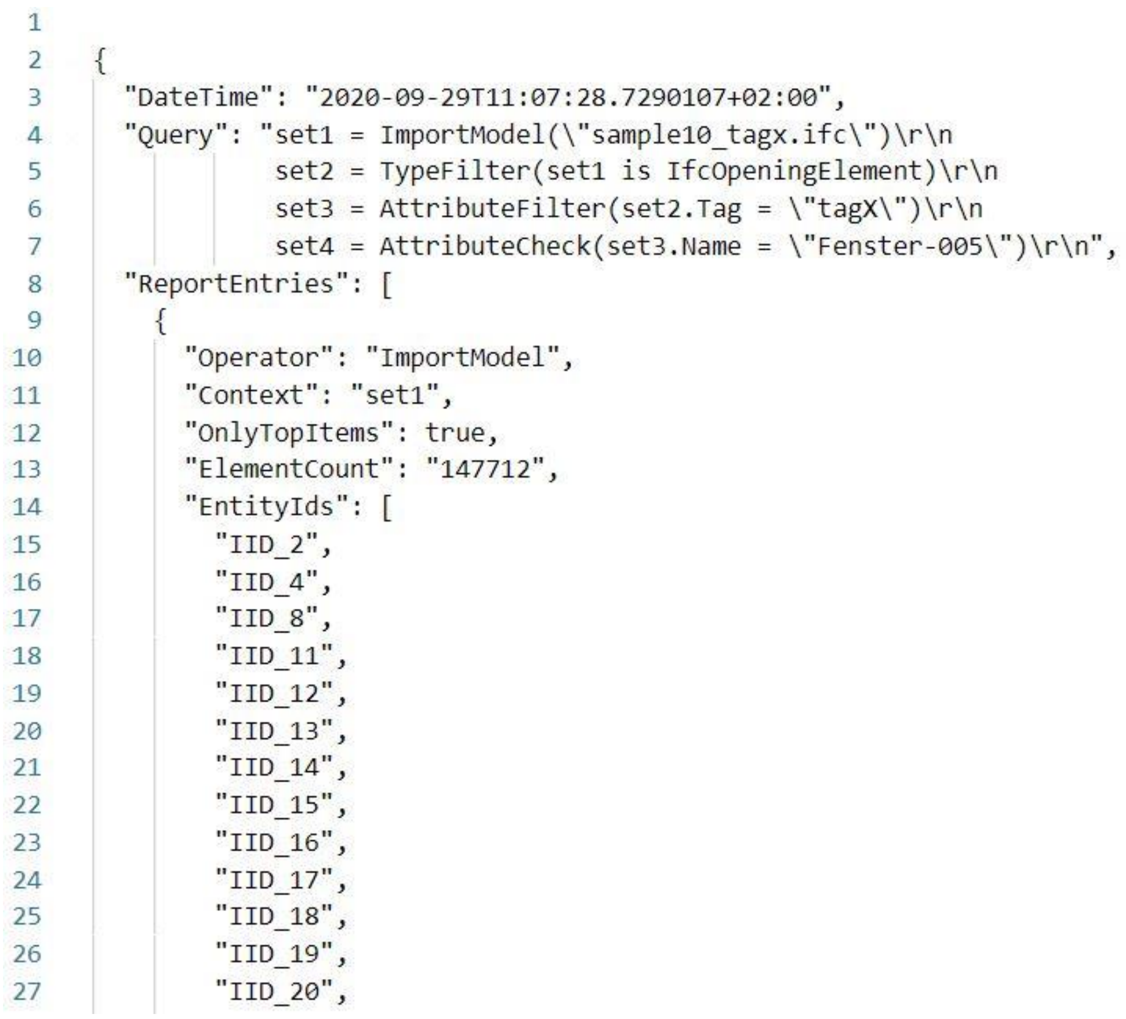
```
        </ConceptTemplate>
</Templates>

<Views>
    <ModelView uuid="72dad5df-6f61-49f2-ba8c-baccf24a6ce5" name="Sensor signal view"
    applicableSchema="IFC4" code="Sensor">
          <Definitions>
            <Definition>
              <Body lang="en"><![CDATA[ModelView for mvdXML 1.1 documentation.]]></Body>
            </Definition>
          </Definitions>
           <ExchangeRequirements>
               <ExchangeRequirement uuid="ae70f764-938b-4cf7-9814-c29a47f56b0e"
               name="Distribution signal" code="ERM1" applicability="export">
                   <Definitions>
                     <Definition>
                       <Body lang="en">
                         <![CDATA[Simple example for checking sensor elements to always submit signals.]]>
                       </Body>
                     </Definition>
                  </Definitions>
               </ExchangeRequirement>
           </ExchangeRequirements>
           <Roots>
               <ConceptRoot uuid="8b949664-a5df-4bfc-922c-4a486c41d756" name="Sensor"
                applicableRootEntity="IfcSensor">
                   <Definitions>
                     <Definition>
                       <Body>
                         <![CDATA[<p>A sensor is a device that measures a physical quantity and converts it
                         into a signal which can be read by an observer or by an instrument.</p>]]>
                       </Body>
                     </Definition>
                   </Definitions>
                   <Concepts>
                      <Concept uuid="a4fa348c-a025-4a02-abfd-c42fd0901540" name="Port Assignment">
                       <Definitions>
                         <Definition>
                           <Body lang="en">
                             <![CDATA[Concept to validate that every sensor elements
                                has a port defined that submits signals.]]>
                            </Body>
                          </Definition>
                        </Definitions>
                        <Template ref="bafc93b7-d0e2-42d8-84cf-5da20ee1480a"/>
                        <Requirements>
                           <Requirement exchangeRequirement="ae70f764-938b-4cf7-9814-c29a47f56b0e"
                              requirement="mandatory" applicability="export"/>
                        </Requirements>
                        <TemplateRules>
                              <TemplateRule Parameters="Name[Value]='Output' AND
                              Type[Value]='SIGNAL' AND Flow[Value]='SOURCE'"
                              Description="Transmits signal."/>
```

```
                                  </TemplateRules>
                               </Concept>
                           </Concepts>
                     </ConceptRoot>
                </Roots>
      </ModelView>
</Views>
</mvdXML>
```

Listing A-1 MvdXML example showing the usage of mvdXML for validation purposes (Chipman et al., 2016)

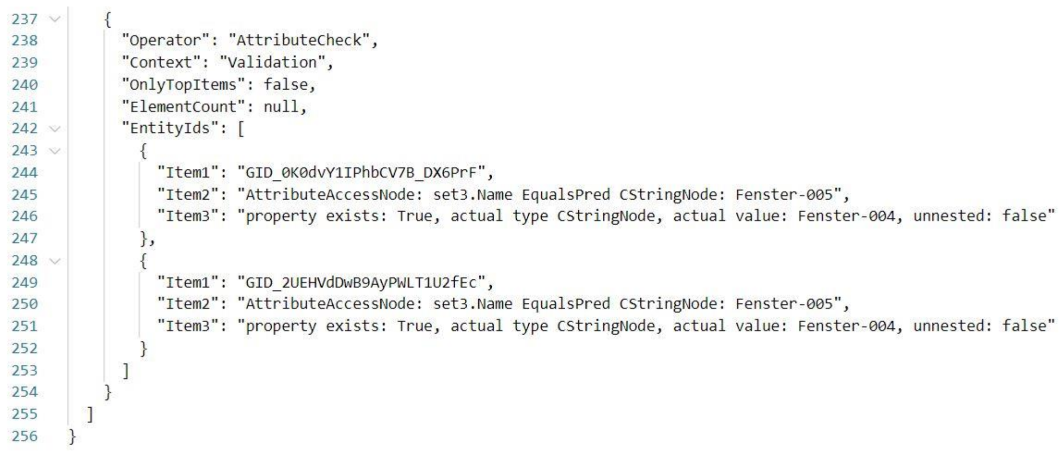
Appendix B

Excerpts of JSON check report file



```

{
  "DateTime": "2020-09-29T11:07:28.7290107+02:00",
  "Query": "set1 = ImportModel(\"sample10_tagx.ifc\")\r\n
           set2 = TypeFilter(set1 is IfcOpeningElement)\r\n
           set3 = AttributeFilter(set2.Tag = \"tagX\")\r\n
           set4 = AttributeCheck(set3.Name = \"Fenster-005\")\r\n",
  "ReportEntries": [
    {
      "Operator": "ImportModel",
      "Context": "set1",
      "OnlyTopItems": true,
      "ElementCount": "147712",
      "EntityIds": [
        "IID_2",
        "IID_4",
        "IID_8",
        "IID_11",
        "IID_12",
        "IID_13",
        "IID_14",
        "IID_15",
        "IID_16",
        "IID_17",
        "IID_18",
        "IID_19",
        "IID_20",
```

Listing B-1 The head of the created JSON check report contains the query. In the use case instances of `IfcOpeningElement` with an attribute value ‘tagX’ of the attribute `Tag` are checked if their attribute `Name` has the value “Fenster-005”. The Check report starts with an enumeration of all entities in the model imported by the ImportModel



```
237      {
238        "Operator": "AttributeCheck",
239        "Context": "Validation",
240        "OnlyTopItems": false,
241        "ElementCount": null,
242        "EntityIds": [
243          {
244            "Item1": "GID_0K0dvY1IPhbCV7B_DX6PrF",
245            "Item2": "AttributeAccessNode: set3.Name EqualsPred CStringNode: Fenster-005",
246            "Item3": "property exists: True, actual type CStringNode, actual value: Fenster-004, unnested: false"
247          },
248          {
249            "Item1": "GID_2UEHVdDwB9AyPWLT1U2fEc",
250            "Item2": "AttributeAccessNode: set3.Name EqualsPred CStringNode: Fenster-005",
251            "Item3": "property exists: True, actual type CStringNode, actual value: Fenster-004, unnested: false"
252          }
253        ]
254      }
255    ]
256  }
```

Listing B-2 The check report end part contains the check result. The use case contains two instances that failed the check. For both failed elements, the GUID (Item1 in line 244 and line 249), the predicate (Item2 in line 245 and line 250) and the reason for the error (Item3 in line 246 and 251) are listed

Erklärung

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

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

München, 29. September 2020

Vorname Nachname

Felix Sirtl

[redacted]