TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology, Chair of Software Technology

Master Thesis

Improving Extensibility and Maintainability of Industry Foundation Classes with Role-oriented Modeling

Martin Klaude

Martin.Klaude@mailbox.tu-dresden.de
Matriculation number: 3965993
Matriculation year: 2018

to achieve the academic degree

Master of Science (M.Sc.)

Supervisor
Dr.-Ing. Sebastian Götz
Supervising professor
Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted on: October 20, 2020

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Improving Extensibility and Maintainability of Industry Foundation Classes with Role-oriented Modeling* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, October 20, 2020

Martin Klaude

Acknowledgments

I will keep this section as short as possible, but I cannot avoid thanking the people who accompanied me through working on this thesis.

First of all, I want to thank my girlfriend Anna, who was also assigned involuntarily to be my proofreader. Thanks for all the love, the care, the strength, and everything else you have given me from the first days of my studies until the finalization of this thesis, which has not been an easy task sometimes.

After that, I want to thank my best friends Christian, Nick, and Clemens, who helped me "switching off" from time to time by playing games, chatting, and simply making after work nights enjoyable.

Furthermore, I want to acknowledge the GRAPHISOFT Education Team, respectively Bianca Uremovic, who accredited me with a free student license for ArchiCAD despite my course of studies.

Last but not least, I want to thank my supervisor Sebastian Götz, who introduced me to the interesting topic, helped me develop crucial ideas, and often brought my thoughts together.

—

Thanks
Martin

Abstract

Nowadays, digitalization supports and even improves more and more areas such as education and healthcare. Actually, areas like the building industry benefit from those advantages as well. Pencil drawings have been replaced by feature-rich 3D models with the help of computer-aided design (CAD) software. Moreover, models of buildings became increasingly "smarter" by appending additional information – which is widely known as Building Information Modeling (BIM). Yet, the most-used data modeling standard – Industry Foundation Classes (IFC) – has shortcomings regarding **maintainablity** and **extensibility**.

Therefore, this thesis focuses on improving these aspects with the help of **role-oriented modeling**. A motivating introduction will mark the beginning by familiarizing the idea of BIM, proposing the methodology and the research questions for this thesis, and elaborating on the *status quo*. Afterwards, a deeper understanding of IFC and its core problems will set the basis for the development of a solution to the identified deficiencies. Prior to that, the basics in role-oriented modeling will be explained. Consequently, the developed role-oriented solution – namely Industry Foundation Classes with Roles (IFC-R) – will be introduced, followed by a comparison of IFC and IFC-R in order to prove its effects. This will be supported by an evaluation of the comparison, which leads to the conclusion of this thesis and a brief outlook for future research.

Acronyms

ADED	analysis, design, evaluation and diffusion
AEC/FM	architecture, engineering, construction, and facilities management
BIM	Building Information Modeling
CAD	computer-aided design
CBO	Coupling Between Object Classes
CC	Cyclomatic Complexity
CFC	Control Flow Complexity
CFG	control flow graph
CROI	Compartment Role Object Instances
CROM	Compartment Role Object Models
DMU	Data Model Understandability
EF	Environmental Factors
EMF	Eclipse Modeling Framework
ETL	Epsilon Transformation Language
IFC	Industry Foundation Classes
IFC-R	Industry Foundation Classes with Roles
ISO	International Organization for Standardization
M2M	model-to-model
MDSE	model-driven software engineering
OOP	object-oriented programming
RE	Runtime Efficiency
SLOC	Source Lines of Code
STEP	Standard for the Exchange of Product Model Data
TCF	Technical Complexity Factors
UAW	Unadjusted Actor Weight
UCP	Use Case Points
UUCP	Unadjusted Use Case Points
UUCW	Unadjusted Use Case Weight
WC	Workflow Complexity

List of Figures

List of Tables

Contents

1. Introduction

During the past years, the usage of software as a tool, e.g. using an integrated development environment (IDE) for developing code, has become natural for most people in the field of software engineering. However, not every area of life takes full advantage of helpful tools for day to day tasks and challenges, yet. Fortunately, digitalization has been progressively enhancing this circumstance in many fields like education, healthcare and various industries. For example, drawing architectural blueprints was already improved in the mid 1960s due to computer-aided design (CAD) software. However, fine-grained distinctions in drawings, e.g. if a line represents a door or wall, remain difficult even in a digital form. That is why, a shift in the world of the building industry was inevitable. The so called Building Information Modeling (BIM) approach emerged [Bor+15b] and along with it a common standard named Industry Foundation Classes (IFC) to model data properly.

Despite being a standard for creating and sharing models, practical usage and research on IFC have revealed shortcomings regarding maintainability and extensibility [Bor+15a; Zhi+11; RFM13; Mot+16]. Therefore, this thesis focuses on addressing these problems by analyzing them and improving the issues by utilizing role-oriented modeling in an approach called Industry Foundation Classes with Roles (IFC-R).

BIM has been developed in the field of civil engineering informatics since the 1990s and tries to incorporate different stakeholders at each stage in the life of a building. Starting from the planning phase over constructing and maintaining the facility until its demolition, every bit of information and data will be stored ideally in one model to reduce errors and improve productivity [Bor+15b]. A common problem, known to anyone who has worked in projects with different stakeholders before, is that each of them will probably use their own tools and data structures. This leads to complications with respect to interoperability resulting in misconceptions, information loss, and decelerated processes.

In order to address these issues, a consortium called *buildingSMART* aims at improving and standardizing the collaboration in the building industry with respect to the idea of BIM. One big achievement of this attempt is the data modeling standard Industry Foundation Classes, which has become the *de facto* model for working and collaborating in the architecture, engineering, construction, and facilities management (AEC/FM) community. To illustrate working with IFC in the context of BIM, figure 1.1 shows an application by *apstex*[1] demonstrating the IFC model of a simple house as well as the stored information.

[1]The webpage of *apstex* can be found here: www.apstex.com

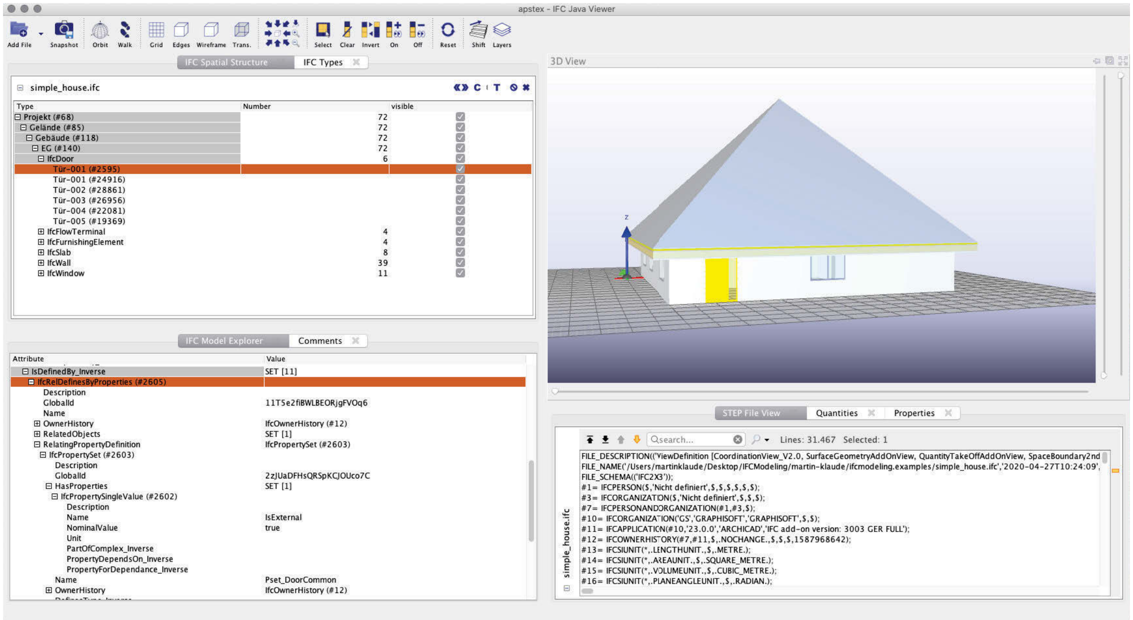


Figure 1.1.: Screenshot of IFC Java Viewer by *apstex* showing a 3D view of a house modeled with IFC

Since Industry Foundation Classes describe models, this approach became interesting to the model-driven software engineering (MDSE) community, which aims at analyzing and understanding the underlying concepts of IFC. For example, Götz et al. have conducted a systematic literature review of over 90 papers (published since 2008) in [Göt+19]. This review showed that the AEC/FM community prefers the technology stack from *buildingSMART*, which means the IFC data model, instead of relying on methods and typical technologies from the MDSE community, such as UML or EMF (Eclipse Modeling Framework) [Göt+19]. This implies that the used technologies and tools are more suitable for AEC/FM contexts, or that there is a lack of awareness concerning the existence of common MDSE methods and technologies or how to use them.

In short, the IFC standard contains shortcomings from the software engineering point of view, which also applies to the resulting models. Since this thesis concentrates on these issues, the following paragraphs will further outline (a) the necessary questions to be answered and (b) the selected approach attempting an improvement of IFC considering the *Status Quo*.

Methodology and Research Questions In order to structure the thesis and to define proper research questions, the applied research process will be introduced, followed by further elaboration on the research questions to be answered.

IFC is an industrial standard used by practitioners in the AEC/FM community, which is why this thesis follows a technical approach resulting in a newly developed solution. Therefore, the ADED (analysis, design, evaluation and diffusion) research process by Österle and Otto [ÖO09] is the method of choice.

In practice that means:

(1) analyzing the problems of IFC in depth,
(2) designing a new approach improving the found shortcomings,
(3) evaluating the newly developed solution with the help of prototypes and, finally,
(4) diffusing the results of the analysis and evaluation in this thesis and the corresponding documentation.

Accordingly, a sufficient problem **analysis** marks the beginning. As stated above, the IFC standard shows weaknesses regarding maintainability and extensibility. These result in workarounds utilizing available mechanisms to enhance the expressivity of the models. However, this circumstance does not fully contribute to the idea of BIM because the full information modeled is still not available immediately. Examining the workaround mechanisms with respect to practical usage and research, as well as understanding IFC and the underlying metamodel language EXPRESS is, consequently, necessary to identify potential aspects for improvements. This leads to the first research question of this thesis, which will be subject of chapter 2:

(RQ1) What are the (core) issues of IFC and how can they be improved?

Secondly, taking the fundamental issues under consideration, the improvement of IFC should not be another workaround, hence, a more sophisticated solution away from the available mechanisms has to be **designed**. However, the understanding of the selected approaches to improve IFC demands a brief overview of possible solutions. So, the corresponding metadata architecture needs a bit more elaboration.

This thesis will be based on the general architecture given by the Object Management Group (OMG) called Meta Object Facility (MOF) [OMG16] and IFC will be organized according to Götz et al. into the levels M3 to M0 [Göt+19]. EXPRESS will be located at level M3 representing the highest level of abstraction because it is able to describe itself and it has been initially used to specify IFC. The Industry Foundation Classes will reside on M2, i.e. the classification of elements like `IfcObject` and `IfcDoor`. Specific *occurrences* such as the entrance door of a building being described by an `IfcDoor` will be located on M1 and runtime objects realizing the IFC data model in an application will be considered on M0 [Göt+19]. A summary of the metadata architecture compared to UML is listed in table 1.1.

Table 1.1.: Metadata architecture of IFC compared to UML

	Classification of IFC	Classification of UML
M3	EXPRESS	MOF
M2	Industry Foundation Classes specification (e.g. definition of `IfcDoor`)	UML (general concepts like `Class`, `Attribute` and `Method`)
M1	*Occurrences* (a specific `IfcDoor` having a `GlobalId`, `Name`, `IfcPropertySets`, etc.)	User-defined UML diagrams (e.g. a class diagram)
M0	Runtime objects in an application	Real objects modeled using UML (e.g. a real door in a house)

The previously mentioned workarounds reside on M1. A proper solution, however, would be to lift those mechanisms on M2 or to have a more mature object-oriented approach.

Adopting the former could be a solution using multi-level modeling [AK01] as mentioned by Götz et al. [Göt+19]. With respect to the latter, an approach with role-oriented modeling could be possible as well. Both are promising approaches, yet, this thesis will concentrate on a solution employing the nature of roles. This constitutes to the relatively intuitive concepts of roles which appear to be easier to grasp by the AEC/FM community than high-level concepts like *Deep Instantiation* [AK01] or *MLT* [CAG16]. For example, a class `Person` playing the role `Employee` while being at work but discarding that role and taking a new one called `Parent` after work while jumping around with the kids will more likely be understood than multiple levels of instantiations [AK01]. The same applies to theories involving concepts like the powertype pattern [CAG16]. Nevertheless, these promising approaches should be considered in future work especially for improving IFC from scratch.

Hence, the developed solution is based on a formal role-modeling language called Compartment Role Object Models (CROM) [Küh+15] and will be labeled Industry Foundation Classes with Roles (IFC-R) from now on. Therefore, the second research question for this thesis reads as follows:

(RQ2) Is role-oriented modeling a solution for the identified deficiencies of IFC?

The main goal of IFC-R is to improve the extensibility and maintainability of IFC models especially for later stages in the lifecycle, e.g. the facility management. For that reason, the primary contributions of IFC-R are (a) a CROM metamodel of essential parts of IFC, (b) a model transformation (`IFC2CROM`/`IFC2CROI`) and (c) a prototypical implementation employing the models, which will be further elaborated in chapter 3.

Lastly, in order to review whether IFC-R contributes to the desired improvements, an empiric evaluation follows in chapter 4, leading to the final conclusion completing the ADED research process. However, to motivate the need to analyze IFC in depth and to identify aspects for improvement, the *Status Quo* in practice and research will be the subject of the next paragraph.

The Status Quo The systematic literature review by Götz et al. revealed that "IFC allows typing objects decoupled from the inheritance hierarchy" [Göt+19] and briefly introduced one mechanism of IFC to achieve extensibility, namely creating orthogonal classifications through **`IfcTypeObjects`**. As mentioned in the review, this so called powertype pattern is "a restricted version of multi-level modeling" [Göt+19] and needs more research to cope with the elaborated approach of BIM. Borgo et al. identified this issue as well while analyzing IFC with respect to creating a corresponding OWL (Web Ontology Language) in [Bor+15a].

Furthermore, adding information to a model by means of adding properties via **`IfcPropertySets`** has been identified as the second main approach to extend IFC models. Zhiliang et al. examined the possible solutions for extensions with respect to cost estimations during construction especially for China [Zhi+11]. They considered the given mechanisms for extensibility mentioned by Weise, Liebich, and Wix in [WLW09]. These are in particular extending the schema definition itself, defining new elements using proxies, or employing property sets [WLW09]. As highlighted by Zhiliang et al. and Weise, Liebich, and Wix, extending IFC itself usually involves many experts discussing each proposal for addition and, therefore, takes a lot of time [Zhi+11; WLW09]. This circumstance has also been mentioned by Rio, Ferreira, and Martins in [RFM13] and Motamedi et al. in [Mot+16] and contributes to the application of `IfcPropertySets` to serve the purpose, though, this brings along new drawbacks like the need to agree on the meaning and usage of the newly added features.

An implication is, according to Zhiliang et al., that these insufficiencies will be fixed by extracting the needed data from the IFC model and providing the additional data in an extra application [Zhi+11]. An example of this implication is illustrated in the paper [RFM13] by Rio, Ferreira, and Martins, in which the authors edited the IFC model in a simple text editor because the given mechanisms for extensibility were not sufficient. In the end, such approaches would create an extensive landscape of tools, each with its own data model, which would also be contrary to the overall idea of BIM.

Besides examining tools that apply IFC models, a certain amount of research must also address the CAD tools for creating the building models. This includes tools like *ArchiCAD* and *Revit*, which, unfortunately, also contribute to the misuse of the available mechanisms. *ArchiCAD*, for example, stores application-specific properties, e.g. if the 2D fixpoint of a visible object should be displayed in 3D. Such unnecessary properties will not be exported if configured properly, but configuration errors are common since configuring such an export is tedious and usually not intuitive. *Revit's* export for user defined property sets, for example, is based on a complex text file and the basic concepts of IFC are not mapped one-to-one, which is why it might be misleading [Hoo19; Mou19].

Improving the default tools used in industry, like *ArchiCAD* and *Revit*, or changing a standard like IFC or its underlying metamodel exceeds the possibility of this thesis. Therefore, the motivation for this work will consider an application with two use cases. One of them reflects an aspect of cost estimation during a construction process and the other one focuses on a facet of the facility management of a building. Both rely on an existing IFC model employing the developed approach IFC-R.

According to the above-mentioned ADED process, the following parts will be structured as follows: Chapter 2 deepens the understanding of concepts and the structure of IFC. It also analyzes EXPRESS as a potential cause for misconceptions during the development of IFC, which tackles the first research question. Chapter 3 covers selected basics in role-oriented modeling, introduces the IFC-R approach, and elaborates on a prototypical implementation applying IFC-R. Afterwards, the empiric evaluation of the developed data models in chapter 4 will tackle the second research question. Chapter 5 concludes this thesis and covers potential aspects of future research.

2. Understanding Industry Foundation Classes (IFC)

As this thesis aims at improving IFC, it is crucial to begin with its general structure and fundamental concepts. This in-depth analysis contributes to the revelation of the core issues regarding extensibility and maintainability; furthermore, examining these aspects will help answer the first research question.

Hence, discussing the foundations of IFC, will be subject to the first section. After that, section 2.2 will focus on EXPRESS, which is the underlying modeling language of IFC. In this way, it should be analyzed whether EXPRESS is a main driver of the deficiencies. Finally, section 2.3 will elaborate on the main problems of IFC regarding its core concepts.

2.1. Structure and Fundamental Concepts of IFC

Since the mid 1960s, the building industry takes recourse to digitalization. From that moment on, technologies such as computer-aided design (CAD) have improved the quality and efficiency in this area long before Industry Foundation Classes (IFC). Therefore, it is worth mentioning the roots from which IFC originated – namely Building Information Modeling.

Building Information Modeling (BIM) To cite from the handbook of the “father of BIM” [Eas+11, p. xiii] – Charles “Chuck” M. Eastman: “Building Information Modeling (BIM) is one of the most promising developments in the architecture, engineering and construction (AEC) industries.” [Eas+11, p. 1]. The first prototype of BIM dates back to 1975 under the name of “Building Description System” [Eas+11, p. xi]. However, the first known usage of the term “Building Modeling” in the sense of Building Information Modeling dates back to 1986 to the title of a paper by Robert Aish [Ais86; Eas+11, p. xii]. The actual term “Building Information Modeling” then first appeared in Van Nederveen and Tolman’s paper from 1992 [VT92; Eas+11, p. xii] and marked an important milestone in its history. All in all, it took almost 25 years from the initial prototype to a wide propagation of the ideas and concepts, which is exemplary for the development process of new technologies in academia.

Since Eastman has started working on his prototype, he has always clearly defined what BIM is and what it is not. On that note, models that only contain 3D information without further definitions by means of attributes do not employ BIM technologies [Eas+11, p. 15].

Classical BIM technologies emerged due to rapid enhancements in CAD software for 2D and 3D drawings and in virtue of inefficiencies in the traditional approaches in the building industry [Eas+11, pp. 1 sqq.]. These former drawings evolved to rich information models based on the ability to add more data, which can then be utilized as a steady basis for the building process. Therefore, BIM is defined "as a modeling technology and associated set of processes to produce, communicate, and analyze *building models*." by Eastman et al. [Eas+11, p. 13].

More importantly, the definition of BIM does not only apply to the construction phase but can also be utilized throughout the entire lifecycle of a building. To be precise, the benefits of BIM reach from pre-construction, like the initial conception, over building design and construction until the post-construction, e.g. facility management, and even the demolition of a building [Eas+11, pp. 16 sqq.]. The mentioned benefits are manifold. If the owner of a building wants an additional entrance to a room, the architect is easily able to change the model accordingly. Such changes, in turn, can trigger a revaluation with respect to cost estimations. Furthermore, other stakeholders can immediately see the adjustments in order to, for example, intervene because the additional entrance has been added to a load-bearing wall resulting in problems with the structure.

Chuck Eastman and all other BIM-contributors started to revolutionize the building industry – as it is known today – with their ideas and concepts for almost half a decade. However, they have not defined or developed specific tools nor technologies. In simplified terms, BIM is an idea or a "guideline" on how to increase the expressiveness of models in order to improve the analysis of information and the collaboration during the lifecycle of a building. As a consequence, not every CAD software or system can be denoted as a BIM application. The main property is defined as "object-based parametric modeling" [Eas+11, p. 25] – generally known as object-oriented modeling in computer science – which enables users to further define their objects by means of properties. That is why applications which offer these functionalities, like *ArchiCAD* and *Revit*, can be denoted as BIM tools. The impact of such tools on the problems of IFC will be discussed in section 2.3.

Building Information Modeling, on the one hand, relies on applications enabling their users to enrich models with information. On the other hand, BIM depends on interoperability. That means being able to share these models without information loss or other problems is crucial for a collaboration. As a result, data models – which have been developed and used in different industries since the 1980s – have also been established with respect to BIM [Eas+11, p. 65]. Both, the International Organization for Standardization (ISO) and the industry have developed such models for data and information exchange [Eas+11, p. 65]. The result of this are many different data models for various industries – obviously, the data model for building product data is called Industry Foundation Classes (IFC).

2.1.1. Organization and Architecture of IFC

IFC was developed and has been maintained by *buildingSMART International* – formerly known as International Alliance for Interoperability (IAI) – since 1994 [LK12]. They describe themselves as "the worldwide authority driving the digital transformation of the built asset environment" [bui20] and their community has members in many countries such as China, the United States, and Germany. Furthermore, *buildingSMART* promotes the ideas and concepts of BIM with their enhanced collaboration initiative called *openBIM* in which IFC is a main driver for interoperability [bui20]. Besides the data model, *buildingSMART* provides standardized terms, described in the *International Framework for Dictionaries* (IFD), and processes defined as methodology under the name *Information Delivery Manual* (IDM). Figure 2.1 illustrates this "triangle of standards", which forms the basis of *openBIM*.

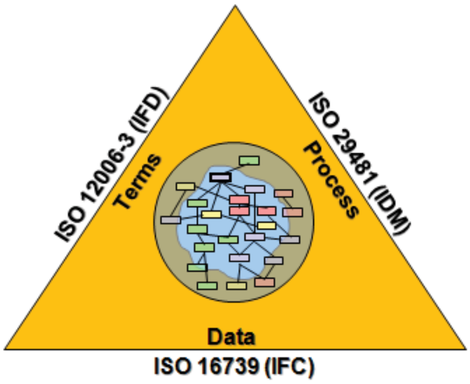


Figure 2.1.: Organization of the standards in *openBIM* [bui18b]

IFC itself is an open ISO standard certified in *ISO 16739-1* and specifies "a data schema and an exchange file format structure" [ISO18]. Currently there are three official data schema versions[1], namely *IFC4.1*[2], *IFC4 ADD2 TC1*[2] and *IFC2x3 TC1*[2], which are defined as an EXPRESS data schema and a XSD (XML Schema Definition) [ISO18]. EXPRESS is also certified by the *ISO* in *ISO 10303-11* and defines "a formal information requirements specification language" [ISO04, p. xii]. For the exchange of data, the standard defines a clear text encoding certified in *ISO 10303-21* and is generally known as Standard for the Exchange of Product Model Data (STEP) [ISO16]. Another more simple way of sharing data is the usage of XML files, however, this thesis concentrates on schema definitions that were created by using EXPRESS – which will be further discussed in section 2.2 – and stored in STEP files.

The IFC specification is divided into four "conceptual layers", in which each layer covers different schemes starting from the definition of general concepts up to specializations of various domains. Figure 2.2 illustrates this so called "data schema architecture" [bui18c].

[1] Unless otherwise stated, IFC will be analyzed in its latest official version, which is *IFC4.1*.

[2] For a complete list of IFC specifications, please see the IFC Specifications Database.

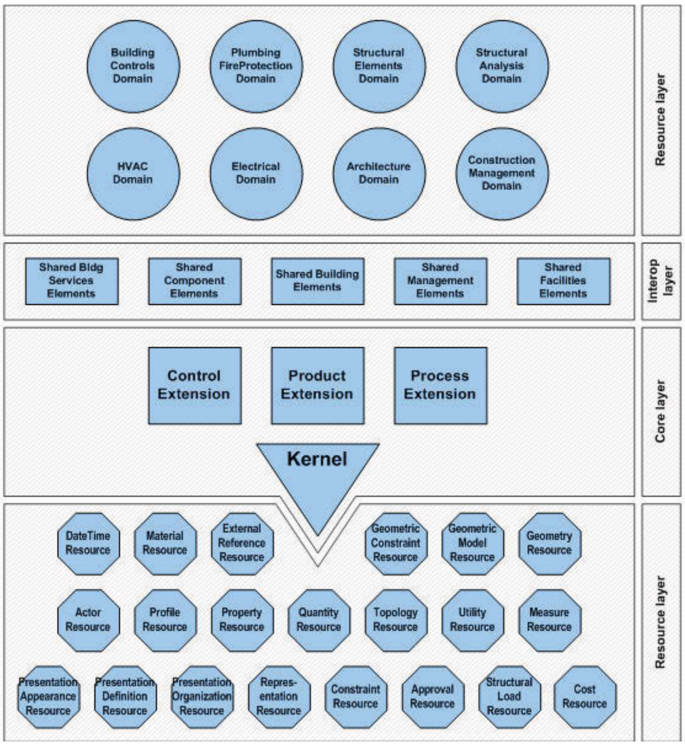


Figure 2.2.: Layered data schema architecture of IFC [bui18c]

The bottom layer, namely the **Resource Layer**, defines data structures which support the overlying layers. For example, the schema definition `IfcGeometryResource` defines types, entities and functions, such as `IfcCartesianPoint`, which are used for geometric representations. Another example is the definition of `IfcPropertyResources`, which defines the `IfcProperty` entity in every form. Such entities have no concept of identity and, therefore, can only exist in relation with entities from the other layers [bui18a].

The second layer – the **Core Layer** – combines the kernel schema definition and the most general core types and entities. To be precise, `IfcKernel` defines the root object `IfcRoot`, which is the most abstract entity and the common supertype of most of the other IFC elements, such as an `IfcDoor`. This sub-supertype concept adds an identity to each object, which enables them to be used independently in contrast to entities defined in the Resource Layer. Listing 2.1 shows the attributes defined for `IfcRoot` in EXPRESS. Beyond that, the kernel contains the schema definitions for objects (`IfcObjectDefinition`), relationships (`IfcRelationship`), properties (`IfcPropertyDefinition`), and core extensions like the definitions for products, which could be building and furnishing elements [bui18a].

Listing 2.1: EXPRESS specification of `IfcRoot` according to *IFC4.1*

```
ENTITY IfcRoot
  ABSTRACT SUPERTYPE OF(ONEOF(IfcObjectDefinition, IfcPropertyDefinition, IfcRelationship))
    ;
    GlobalId : IfcGloballyUniqueId;
    OwnerHistory : OPTIONAL IfcOwnerHistory;
    Name : OPTIONAL IfcLabel;
    Description : OPTIONAL IfcText;
  UNIQUE
    UR1 : GlobalId;
END_ENTITY;
```

The **Interoperability Layer** is the next layer, and it defines necessary schemes for the inter-domain exchange, i.e. for the sharing of information which might be needed by multiple domains. For example, `IfcSharedBldgElements` defines – besides types and entities – property and quantity sets, which support the description of building elements such as `IfcDoor`, `IfcDoorType` and `Pset_DoorCommon` [bui18a].

The top and last layer covers intra-domain exchange between different industry disciplines; hence, this layer is called **Domain Layer**[3]. The schema definition of `IfcSharedBldgServiceElements` in the underlying layer, for example, defined basic blocks for interoperability between different domains. These basic blocks will be further specialized in this layer, e.g. does `IfcElectricalDomain` extend these blocks with respect to various kinds of electrical systems and their connections such as cables [bui18a].

This specific, layered architecture has advantages and disadvantages. One of the biggest advantages is the attempt to reuse as many concepts, entities, and types as possible, which is supported by defining the schemes in different layers. To be precise, the sub-supertype concept introduces more abstract objects in the lower layers which can then be further defined in the upper layers relying on the same parent objects. This kind of reuse reinforces the semantic relation between the schemes and their objects if well-used. On top of that, such an architecture is further stackable, and changes in upper layers only have minor effects on the lower layers.

Considering the layered architecture vice versa, yet, reveals one of its biggest disadvantages because changes in lower layers can have a high impact on the upper layers, if not well-separated nor structured. In addition to that, the promising concept of reusability is hampered by an inconsistent utilization throughout each layer. As a result, extensibility is reduced which provokes the workarounds mentioned in the introduction.

The assumption here is that the separation into several layers was not intended in the first drafts for the Industry Foundation Classes. The initial design was a minimalist approach in order to enhance the data exchange [LK12]. To some extent, it can be argued that the lower layers still follow that minimalist approach because they offer only basic concepts and attributes. However, the structure of IFC is not as loosely coupled as striven for, considering the intended usage and implementation of the entire data model [LK12]. This could be explained by the fact that IFC was not designed newly from scratch. The first release reused many objects and types that were already defined in the *ISO* STEP standard [LK12]. Additionally, missing ontologies from the beginning and the struggle of market share with proprietary data models of BIM applications like *AutoCAD* by *Autodesk* have contributed to this circumstance [LK12]. Altogether, the presented issues could have developed over the years due to different problems such as standardization issues during the history of IFC. Although interesting, these kinds of problems will not be discussed in this thesis; for further reading, Laakso and Kiviniemi have reviewed the standardization history of IFC in depth in [LK12].

After comprehending the structure and architecture of IFC, the fundamental concepts, which will be examined in the upcoming section, can be classified more easily. To be precise, the core concepts of IFC are located in the **Kernel** (figure 2.2), which means they reside in the second layer – the core layer. As a consequence, improving these concepts is not

[3]The top layer in figure 2.2 was probably named incorrectly. Instead of “Resource layer” it should be named “Domain layer” as the documentation for IFC4 shows (see IFC4 Documentation: Introduction).

trivial because changes in the second layer can have a high impact on the upper layers. However, before improving these concepts, the problem analysis will – for now – focus on their examination.

2.1.2. Examination of the Concepts

It has been argued that the structure of IFC with respect to the resulting models is a problem. Therefore, an examination of real world examples with respect to the fundamental concepts is vital in order to find potential aspects for improvements. As a starting point, an excerpt of an IFC data model saved as a STEP file is shown in listing 2.2.

Listing 2.2: Excerpt of an IFC data model (*IFC2x3*) stored as a STEP file

```
1  ISO-10303-21;
2  ...
3  DATA;
4  #1= IFCPERSON($,'Nicht definiert',$,$,$,$,$,$);
5  #3= IFCORGANIZATION($,'Nicht definiert',$,$,$);
6  #7= IFCPERSONANDORGANIZATION(#1,#3,$);
7  #10= IFCORGANIZATION('GS','GRAPHISOFT','GRAPHISOFT',$,$);
8  #11= IFCAPPLICATION(#10,'23.0.0','ARCHICAD','IFC add-on version: 3003 GER FULL');
9  #12= IFCOWNERHISTORY(#7,#11,$,.NOCHANGE.,$,$,$,1591516987);
10 ...
11 #19390= IFCDOOR('3oJIvZHwmpIvsDNVlm1TkW',#12,'T\X2\00FC\X0\r-005',$,'DOOR',#19220,#19386,'
       F24D2E63-47AC-334B-9D8D-5DFBF005DBA0',2.01,0.885);
12 #19393= IFCRELFILLSELEMENT('09OsjJN6$XN3l0jyMha2mw',#12,$,$,#19041,#19390);
13 #19397= IFCPROPERTYSINGLEVALUE('IsExternal',$,IFCBOOLEAN(.F.),$);
14 #19398= IFCPROPERTYSET('2rs$EFBtgMCc8K5Wf8ZOUk',#12,'Pset_DoorCommon',$,(#19397));
15 #19400= IFCRELDEFINESBYPROPERTIES('2580df7BGyK45cqU2E8jP2',#12,$,$,(#19390),#19398);
16 ...
17 #20150= IFCDOORSTYLE('0SK$AmNsEXJRS5UDyM_xiv',#12,'T\X2\00FC\X0\r Ol 1-Fl 23',$,$,$,$,'1
       C53F2B0-5F63-A14D-B705-78DF16FBBB39',.SINGLE_SWING_LEFT.,.NOTDEFINED.,.F.,.F.);
18 #20151= IFCRELDEFINESBYTYPE('26p_KrRC4PKJEIyrMYIVIa',#12,$,$,(#19390,#22102,#24937,#28888)
       ,#20150);
19 ...
20 END-ISO-10303-21;
```

Such an STEP file is separated into different parts such as a `HEADER_SECTION` (`HEADER;`) and a `DATA_SECTION` (`DATA;`) according to [ISO16]. The main part of the modeled data can be found in the data section (the start is denoted with `DATA;`) which consists of an almost consecutively numbered list of entity instances [ISO16]. The name of such an entity instance starts with the number sign (#) followed by digits (numbers `0` to `9`) only. For example, one instance of an `IfcDoor` entity is named `#19390` in listing 2.2. The rest of an entry consists of the capitalized entity name as defined by EXPRESS, e.g. `IFCDOOR`, and a list of values containing information about the entity instance, like text or references to other instances that correspond to its EXPRESS definition.

Due to its structure, an STEP file only needs a decent amount of memory to store all the information. This is a positive result of the heavy usage of references, such as in line 15 in listing 2.2, which relates the `IfcDoor` entity instance to the instance of the `IfcPropertySet` defined in line 14. However, the usage of references in that way and the general structure of STEP do not allow a full comprehension of the data schema architecture [LK12]. Although promoted as human-readable, the process of parsing and structuring STEP files is, consequently, necessary in order to read and understand the data models in their entirety [LK12].

The result of such a process has been shown in figure 1.1 with the *IFC Java Viewer* by *apstex* which helped, in addition to the IFC documentation, analyze the upcoming fundamental concepts of IFC.

Object Occurrence and Object Type The definition of "object" is versatile in the context of IFC. It ranges from "all physically tangible items" and "physically existing items" over "conceptual items" up to "processes [...], controls [...], resources [...and...] actors" [bui18a]. Each object is defined according to its relationships, which will be explained later.

Objects are categorized as individual objects – denoted as **object occurrences** – or as **object types**. This "type-ocurrence dichotomy" [Bor+15a] is the foundation for each of the upcoming concepts. A specific instance of an object occurrence or object type can be considered as an instance of a class using object-oriented terminology. However, the object type – despite being an individual entity – can be considered as a superclass regarding inheritance, which will be further discussed in section 2.3.

[Bor+15a] denoted the relation between occurrence and type as "typization" because this relationship can determine the properties that are needed to identify an occurrence as part of a certain type [Bor+15a]. However, the identification does not rely on the instantiated values. This relationship is denoted as "realization" by the authors [Bor+15a]. For instance, occurrences of the `IfcDoorType` are represented as instances of `IfcDoor` [bui18a] because a door would be associated to having certain door lining properties defined by the object type. This ontological analysis of IFC was part of [Bor+15a] and will be further addressed as well.

Object Typing An object occurrence can be defined by the **Object Typing** concept [bui18a]. This concept introduces semantic definitions of types which can be assigned to occurrences. As a result, an object occurrence instance can partly or completely apply the common characteristics, e.g. typical material properties defined by a specific `IfcTypeObject` (see listing 2.3) instance [bui18a].

Listing 2.3: EXPRESS specification of `IfcTypeObject` according to *IFC4.1*

```
ENTITY IfcTypeObject
  SUPERTYPE OF(ONEOF(IfcTypeProcess, IfcTypeProduct, IfcTypeResource))
  SUBTYPE OF (IfcObjectDefinition);
    ApplicableOccurrence : OPTIONAL IfcIdentifier;
    HasPropertySets : OPTIONAL SET [1:?] OF IfcPropertySetDefinition;
  INVERSE
    Types : SET [0:1] OF IfcRelDefinesByType FOR RelatingType;
  WHERE
    NameRequired : EXISTS(SELF\IfcRoot.Name);
    UniquePropertySetNames : (NOT(EXISTS(HasPropertySets))) OR IfcUniquePropertySetNames(
    HasPropertySets);
END_ENTITY;
```

For example, the `IfcDoor` instance `#19390` in listing 2.2 is initially defined by an instance of `IfcDoorStyle`[4] in line 17 and line 18. This subtype of `IfcTypeObject` holds common attributes like `OperationType`, which "defin[es] the general layout and operation of the door type" [bui18a]. On top of that, the `HasPropertySets` attribute from `IfcTypeObject` allows the addition of further common properties to all occurrences of the same type. All attributes and properties, however, can be overridden at each occurrence individually, if needed.

[4] `IfcDoorStyle` from *IFC2x3* has been enhanced and renamed to `IfcDoorType` in *IFC4.1*.

Property Sets The second concept for defining objects is by means of **Property Sets**. This concept applies to occurrences as well as to object types, as mentioned in the previous paragraph. An occurrence or object type can be related to a single or to multiple property sets, each containing at least one property (subtypes of `IfcProperty`) [bui18a]. This can be read off the EXPRESS definition of the entity `IfcPropertySet` as seen in listing 2.4.

Listing 2.4: EXPRESS specification of `IfcPropertySet` according to *IFC4.1*

```
ENTITY IfcPropertySet
  SUBTYPE OF (IfcPropertySetDefinition);
    HasProperties : SET [1:?] OF IfcProperty;
  WHERE
    ExistsName : EXISTS(SELF\IfcRoot.Name);
    UniquePropertyNames : IfcUniquePropertyName(HasProperties);
END_ENTITY;
```

The definition of property sets can be either static, which refers to certain property set entities of the IFC specification, or dynamically extendable [bui18a]. To be precise, the semantic meaning of statically defined property sets is associated with its entity type and properties, e.g. the entity `IfcDoorLiningProperties` that defines characteristics of door linings. In contrast, dynamically extendable means that the general entity `IfcPropertySet` is something like a metamodel which needs further agreement in order to create a semantic meaning [bui18a]. Moreover, these kinds of property sets can have an underlying template for the handling of external libraries.

The concept of **Quantity Sets** is equal to the described approach of property sets, despite that these sets contain quantities which define physical properties of elements. Furthermore, IFC offers a set of predefined property sets, e.g. `Pset_DoorCommon` which defines common properties of doors like the acoustic rating. IFC provides the following naming convention for property and quantity sets that have been defined as part of this specification: "Pset_Xxx"[5] for `IfcPropertySet` and "Qto_Xxx"[5] for `IfcElementQuantity`.

Objectified Relationships As described earlier, the enrichment of "objects" with information relies on relations. That means, each object occurrence or type is defined by its relationships. For example, the entity instance `#19390` from listing 2.2 is further defined by a property set (`IfcRelDefinesByProperties` in line 15) and by an object type (`IfcRelDefinesByType` in line 18). The EXPRESS definitions of those relationships can be seen in listing 2.5 and listing 2.6.

Listing 2.5: EXPRESS specification of `IfcRelDefinesByProperties` according to *IFC4.1*

```
ENTITY IfcRelDefinesByProperties
  SUBTYPE OF (IfcRelDefines);
    RelatedObjects : SET [1:?] OF IfcObjectDefinition;
    RelatingPropertyDefinition : IfcPropertySetDefinitionSelect;
  WHERE
    NoRelatedTypeObject : SIZEOF(QUERY(Types <* SELF\IfcRelDefinesByProperties.
    RelatedObjects | 'IFCKERNEL.IfcTypeObject' IN TYPEOF(Types))) = 0;
END_ENTITY;
```

[5] These prefixes should not be used for property/quantity sets defined outside of this specification [bui18a].

Listing 2.6: EXPRESS specification of `IfcRelDefinesByType` according to *IFC4.1*

```
ENTITY IfcRelDefinesByType
  SUBTYPE OF (IfcRelDefines);
    RelatedObjects : SET [1:?] OF IfcObject;
    RelatingType : IfcTypeObject;
END_ENTITY;
```

In classical object orientation, relations are mostly implicit, which means that objects are related to each other, for example, by associations. Most of the relationships in IFC, however, are defined as so-called **Objectified Relationships**. As shown in the listings, the relationships have their own entities, which allows them to keep specific properties only relevant for the relation at individual instances [bui18a]. This approach has been chosen in order to uncouple the relationship semantics from the objects and to allow separate subtype trees for special handling dedicated to relationships [bui18a].

All things considered, the definition of the term "object" and the concepts of object typing, property sets, and objectified relationships are the fundamentals of IFC regarding object models. This abstract specification is defined in the kernel and will be used – and further specialized – by the upper layers with respect to the AEC/FM context. Figure 2.3 consolidates the relation between the concepts in an example defining a door. These concepts, however, appear inconvenient from the object-oriented modeling point of view, which is why they will be further analyzed in section 2.3.

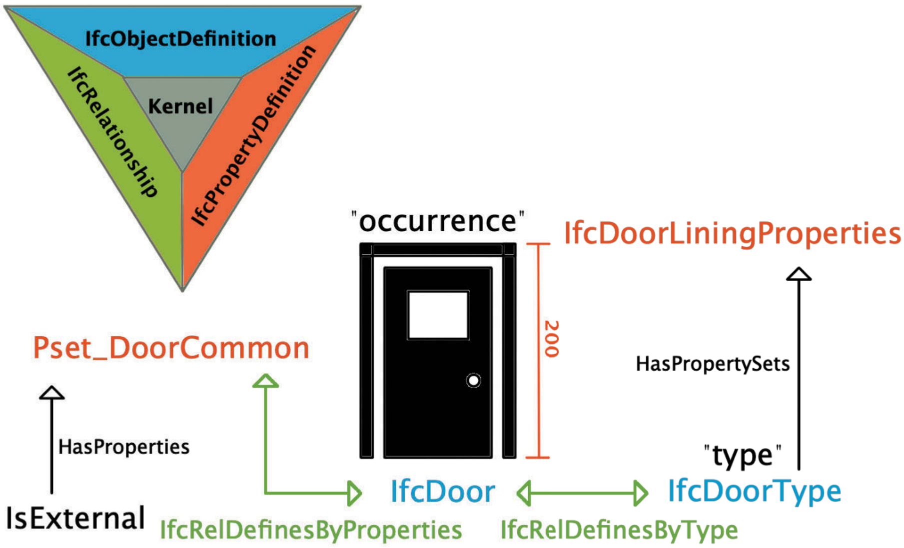


Figure 2.3.: Exemplified relation between the core concepts

To sum it up, Building Information Modeling (BIM) matured from an initial prototype by Chuck Eastman to a game-changing methodology that revolutionized the building industry as it is known today. BIM relies on technologies which support its ideas and concepts – one of them being the need for interoperability. *buildingSMART* followed that need and developed IFC, which is an appropriate approach to exchange building product data. After the examination of the architecture and core concepts of IFC, I will now take a look at the underlying modeling language in order to asses its influence on these fundamentals.

2.2. The Modeling Language EXPRESS

As mentioned above, IFC originated from the *ISO* standard STEP. To be precise, STEP defined standards for several industries such as AEC/FM, which have been reused in the initial release of IFC [LK12]. During the development of STEP – in the 1990s – several existing data modeling languages were evaluated and none of them fulfilled its requirements to a satisfying extent [LK12]. As a result, the modeling language **EXPRESS** was developed as a part of the STEP standard [SW94; LK12]. Although all schema definitions of IFC exist in a modern approach like XSD, EXPRESS is still used to define schemes as well. That is why an assessment of EXPRESS is essential.

The modeling language EXPRESS was released in 1994 by Douglas Schenck, Peter Wilson, and their development team [SW94]. It is defined as "an object-flavored information model specification language" [SW94, p. xxiii], which should fulfill the needs of the STEP standard. EXPRESS defines a family of languages namely (i) the textual language EXPRESS itself, (ii) a graphical language called EXPRESS-G, and (iii) EXPRESS-I, which allows the creation of entity instances with respect to EXPRESS definitions. EXPRESS-G helps comprehend the structure of the data models, but it does not cover all features of EXPRESS [SW94, p. 25]. Hence, it will not be further discussed in this thesis. Similarly, EXPRESS-I will not be assessed as well because it was rather intended for testing than for real usage [SW94, p. 25], and the corresponding *ISO* standard (*ISO 10303-12:1997*) has been withdrawn in 2013 [ISO13].

The EXPRESS language aims at describing characteristics of information which means describing what properties "things" can have, and how they behave and interact with each other [SW94, p. xxiii]. Therefore, it is denoted as **information modeling language**. Schenck and Wilson formulated the definition of an **information model** as follows:

> "*An information model is a formal description of types of ideas, facts and processes which together form a model of a portion of interest of the real world and which provides an explicit set of interpretation rules.* [...]" [SW94, p. 10]

In the core of this definition, EXPRESS – as an information modeling language – enables users to model information by defining attributes and the behavior of "things". Furthermore, the authors and developers of EXPRESS differentiated strictly between data modeling and information modeling – the former is solely designated to be interpreted from a computer system and the latter is not [SW94, p. 10]. However, they stated that "[...] information modeling partakes aspects of both data and Object-Oriented modeling [...]" [SW94, p. 3]. As a result, EXPRESS can be utilized to define objects of the real world in order to exchange their information with respect to either computer-interpretable or human-readable formats. The representation of such object-definitions has already been shown in the EXPRESS specifications of the previous section, e.g. listing 2.3 of the `IfcTypeObject`. Nevertheless, they have not been explained in detail with respect to the building blocks of the language, which is why the upcoming section will cover more details of the EXPRESS language.

2.2.1. Building Blocks of EXPRESS

First of all, the term "thing" needs a redefinition. The starting point – equal to object-oriented modeling – is the question: "What do I want to model?". The "what" refers to real world **objects** most of the time, for example a **particular** door in the specified terminology [SW94, pp. 14 sq.]. In contrast, the **generic** description of such objects, e.g. doors in general, will be denoted as **class**. The same terminology applies to object-oriented programming (OOP), whereas EXPRESS denotes generic classes as **entities** and the particular object as **instance**. Therefore, the first building block of EXPRESS is the categorization of classes by means of entities [SW94, pp. 42, 156 sqq.]; listing 2.7 illustrates the door class in EXPRESS.

Listing 2.7: Door entity defined in EXPRESS

```
ENTITY door
  SUBTYPE OF (product);
END_ENTITY;
```

In addition to the simple definition of a door as an entity, listing 2.7 introduces another important feature of EXPRESS: **generalization**. This concept uses the keywords `SUBTYPE OF`, which marks the entity `product` as the supertype of the entity `door`. This sub-supertype concept [SW94, pp. 42, 86 sqq.] – also known as generalization – is also established and often used in OOP. This way of relating entities creates an inheritance hierarchy. The advantages of this hierarchy can be explained by the example definition of `product` in listing 2.8.

Listing 2.8: Product entity defined in EXPRESS

```
ENTITY product;
    material : STRING;
    price    : REAL;
  WHERE
    price_over_zero : price > 0;
END_ENTITY;
```

As it can bee seen, the entity `product` has two **attributes** and one **constraint** [SW94, pp. 43 sqq., 156 sqq.]. Modeling information means being able to distinguish particular objects. This ability will be achieved by adding attributes and by constraining them, if needed. Besides being the next building blocks of EXPRESS, the attributes and constraints of a supertype will be inherited by all its subtypes, which means an instance of `door` must be made of a certain `material` and must have a specific `price` *greater than zero* because selling products expects profit as well.

The next essential concept of EXPRESS is the **types** construct [SW94, pp. 50 sq., 154 sq.]. Typing has already been mentioned in the context of IFC's core concepts and is crucial for modeling in order to increase the semantic meaning of the models. Therefore, let me revise the `product` entity: an attribute named `price` usually mirrors the (monetary) value of an object and will be defined using the simple data type `REAL` (to represent a decimal) in this example. However, there is no currency attached to the price, so one could pay me with, e.g. 300 apples if the price will be instantiated with a value of `300`. This semantic issue can be resolved by introducing a special type as shown in listing 2.9.

Listing 2.9: Money type defined in EXPRESS

```
TYPE money = REAL;
  WHERE
    over_zero : SELF > 0;
END_TYPE;
```

The type `money` has the same simple data type as the `price` attribute but has an enhanced semantic meaning. Additionally, the constraint of the value being greater than zero can be transferred to the type since this constraint logically belongs to the type rather than the entity (in this example). Another special and important constraint of attributes is the so-called **existence constraint** that is declared by using the `INVERSE` keyword [SW94, pp. 47 sq.].

Obviously, this constraint checks for dependencies regarding the existence of one instance with respect to another – in OOP this is considered as composition. In order to exemplify this concept, the `product` entity will be revised again in listing 2.10 supported by listing 2.11.

Listing 2.10: Product entity revised (EXPRESS)

```
ENTITY product;
    material : STRING;
    price    : money;
    owners   : LIST [0:?] OF owner;
END_ENTITY;
```

Listing 2.11: Owner entity defined in EXPRESS

```
ENTITY owner;
    name : STRING;
  INVERSE
    assigned_to_product : product FOR owners;
END_ENTITY;
```

The examples read as follows: Each time a `product` instance is being sold or resold, the new owner is added to the list of `owners`. This list and each instance of an `owner`, however, is only needed as long as the product exists. That means, if the product gets destroyed, the information of each previous owner is not needed anymore and should be deleted as well as the information about the product. This special constraint is heavily used in the resource layer of IFC because the entities in this layer are not intended to be used without a reference to an entity of the overlying layers.

Last but not least, the relations between entities, types, and attributes need assessment [SW94, pp. 160 sq.]. EXPRESS differentiates between three kinds of relationships:

- "is-a": Relationship between a subtype and a supertype, e.g. `door` "is-a" `product`.
- "is-defined-by": Relationship between an entity and an attribute, e.g. `product` "is-defined-by" its `material`.
- "is-represented-as": Relationship between an attribute and a type, e.g. `price` "is-represented-as" `money`.

All relationships are bidirectional, for example the opposite of `is-defined-by` is obviously "defines" [SW94, p. 160].

As opposed to "is-defined-by"-relationships, relations regarding attributes defined in an `INVERSE` clause semantically do not mirror definition but existence [SW94, p. 161]. These relationships do not further define an entity. Instead, they highlight the fact that a certain entity cannot exist without the referencing entity and should be deleted if this entity is no longer existing.

In short, the basic building blocks of EXPRESS are entities and types. Entities can be further defined by attributes and constraints; types, in turn, increase the semantic meaning of attributes supported by constraints as well. Additionally, the relationships between those building blocks are an essential part in information modeling. Although this introduction into EXPRESS might seem comprehensive, it only covers the necessary parts for the upcoming assessment of EXPRESS with respect to its influence on IFC.

2.2.2. The Influence of EXPRESS on IFC

After the examination of the fundamentals of both IFC and EXPRESS, the influence of the latter on the former can be assessed. Without wanting to anticipate the problems of section 2.3, the issues that will be explained there could be theoretically attributed to a lack of proper usage of object orientation. This raises the questions – especially from the viewpoint of MDSE: Why do IFC models not take recourse to classical object-oriented concepts? Is the underlying modeling language a reason for that? If not, was this a conscious decision with respect to standardization or a lack of knowledge about proper approaches during the development of IFC?

Laakso and Kiviniemi already mentioned that the influence is difficult to assess [LK12]. Furthermore, the first release of IFC was way back in time – almost a quarter of a century ago. For that reason, it can be assumed that more appropriate approaches exist today. For example, Laakso and Kiviniemi mentioned the absence of a proper ontology during the early development of IFC [LK12]. This circumstance has also been discussed by the ontology community, for example in [BVD09; Bor+15a; FRN15].

Beetz, Van Leeuwen, and De Vries developed *IfcOWL*, which can transform an EXPRESS schema into a proper ontology, and they presented their work in [BVD09]. The authors stated several starting points for such a transformation in order to improve interoperability in the building industry. They decided to begin with the aforementioned language constructs and brought up the following issues of EXPRESS regarding a proper ontology [BVD09]: (i) lack of formal rigidness, (ii) limited reuse and interoperability, and (iii) lack of built-in distribution. Furthermore Beetz, Van Leeuwen, and De Vries argued that a careful augmentation will be more suitable for this industry than a complete replacement with a new solution started from scratch [BVD09]. In addition to their work, Borgo et al. analyzed the fundamentals of IFC with respect to EXPRESS in order to develop strategies for a proper transformation, and Farias, Roxin, and Nicolle developed *IfcWoD*, which is now stated as a new ontology based on *IfcOWL* in order to improve certain aspects [FRN15].

Contrary to an ontological approach, this thesis tries to improve IFC from the viewpoint of the modeling concepts (MDSE), specifically the object-oriented modeling. Therefore, it is relevant to compare EXPRESS with object-oriented modeling, respectively OOP. This comparison considers essential differences between EXPRESS and classical object orientation, e.g. an evaluation of the concept of inheritance.

First of all, Schenck and Wilson initially denoted EXPRESS as an "object-flavored" language [SW94, p. xiii] and, consequently, isolated their language from strict object orientation. This describes EXPRESS pretty well, in my opinion. On the one hand, there is sufficient support of typical OOP concepts like **classes** (in the form of entities), **inheritance**, and polymorphic constructs like **aggregation** and **object composition**. On the other hand, EXPRESS considers itself as a pure and static information modeling language, which means it only mirrors the modeled entities with their attributes and relations but does not allow modeling of interactions. In practice that means, entities in EXPRESS do not have **methods** and, therefore, no constructs like **encapsulation** or **delegation** is needed. Although it is possible to define functions and procedures in EXPRESS, these algorithmic constructs can only be utilized in order to validate the constraints of attributes. Furthermore, **instantiation** differs from classical object orientation. Despite the terms "instance" or "occurrence" in EXPRESS and IFC, there is no underlying process of instantiation compared to OOP; however, this is no flaw because EXPRESS is a modeling language and no programming language.

To put it in a nutshell, EXPRESS is a solid language for modeling information and data with respect to real world objects. It is able to define entities with all their attributes which can be further constrained and semantically enriched by means of types. Moreover, the relationships between all building blocks of EXPRESS are strictly separated and comprehensible. However, the language lacks in the ability to represent the behavior and possible interactions of objects. On top of that, EXPRESS has further shortcomings, which have already been mentioned by, for example, Beetz, Van Leeuwen, and De Vries [BVD09]. One of them being a limited acceptance in the engineering community in general.

Aside from all those aspects, the previous question still remains unanswered: Is EXPRESS a reason for the problems of IFC? I am of the opinion that this is **not necessarily** the case because EXPRESS has sufficient constructs for proper modeling. To be precise, the structure and fundamentals explained in section 2.1.1 and 2.1.2 are not completely dictated by EXPRESS. Last but not least, changing the underlying modeling language does not automatically resolve the issues of IFC. This has also been stated by Borgo et al.: "If the change of language helps to improve the system as we have argued, it does not per se lead to the clarification of the IFC conceptualization and ontological coherence." [Bor+15a].

EXPRESS might have some flaws, but these do not solely contribute to the weaknesses of IFC.

2.3. Analysis of Core Issues

Building Information Modeling (BIM) – as introduced in this chapter – is a methodology for dissolving solo work from stakeholders and to improve the needed collaboration in the building industry. This should be achieved by different technology stacks. Moreover, BIM has a vivid definition which technologies contribute to its idea and which do not, e.g. simple geometric representations of a building are far away of being state-of-the-art. Therefore, modelers need to understand the value of information and how to proper model these information, their interconnections, and their properties beyond the scope of a single drawing [Hoo18a]. It is crucial to model data as precisely as possible and to understand interoperability. More precisely, the developed data models must be processable outside the limits of a single application, for example *ArchiCAD* or *Revit* [Hoo18a]. All in all, there are various aspects to which modelers need to pay attention. However, the main focus for this section is: Can modelers employ IFC in order to achieve the needed granularity?

Although EXPRESS is the underlying modeling language, IFC is something like a modeling language itself because it offers modelers a way to express information for building product data. As a consequence, it should fulfill the fundamental attributes of modeling languages as defined by Halpin and Morgan: (i) expressivity, (ii) clarity, and (iii) parsimony [HM08; CAG16]. If a language is able to model all relevant aspects of the real world, which should be represented, then it can be considered **expressive** [CAG16]. On the one hand, IFC is a substantial and extensible standard covering many facets of the AEC/FM area – so, it could be considered expressive. On the other hand, IFC lacks in the other two attributes. Specifically that means, if each model cannot be unambiguously interpreted by any user, then a language does not fulfill the need for **clarity** [CAG16]. Moreover, if a modeler has to represent more information than needed in reality, the language is lacking in **parsimony** [CAG16]. Both attributes are not fulfilled by IFC because the classes still rely on human interpretation as a result of missing formal semantics [Bor+15a].

In consequence, modelers are very likely to resolve this circumstance by means of workaround mechanisms. These workarounds utilize two of the fundamental core concepts – namely **property sets** and **object typing** – which is why these fundamentals will be analyzed further. The relation between these concepts with respect to object-definitions can be recalled in figure 2.4.

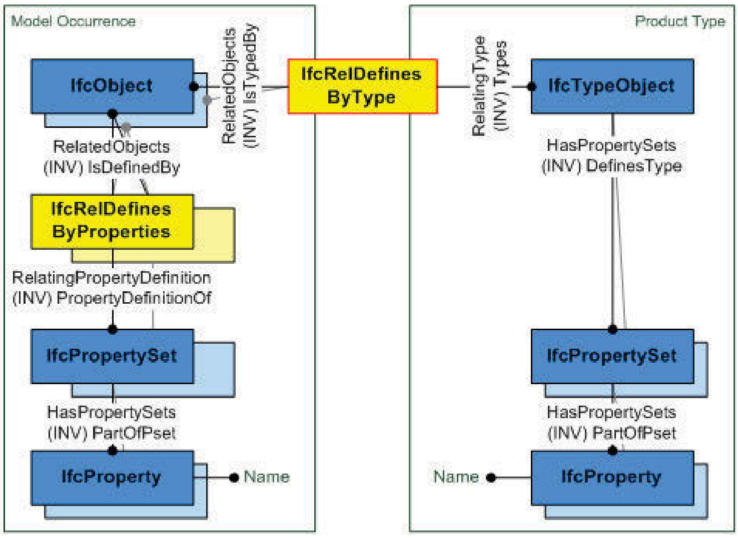


Figure 2.4.: Object-definition by means of types and property sets [bui18a]

2.3.1. Adding properties by means of property sets

Besides its attributes, each sub-entity of `IfcObject`, e.g. `IfcDoor`, can be further defined by means of **properties**. In short, an `IfcProperty` is part of an `IfcPropertySet`. This property set, in turn, is attached to the `IfcObject` through the objectified relationship `IfcRelDefinesByProperties` as shown in figure 2.4. This way of storing information is unusual and slightly more complex compared to classical object-oriented programming.

The upcoming example will expand upon these differences: In practice, the class `IfcDoor` is used to describe and store necessary information about specific doors in a building. This class is able to store predefined attributes such as `OverallHeight` and `OverallWidth`. However, further properties must be stored in the class `IfcPropertySet`; more precisely, each attribute can be stored in an `IfcPropertySingleValue` for example. These property sets further "define" the doors, which is why the `INVERSE` attribute is called `IsDefinedBy`. Figure 2.5 illustrates this example in a UML class diagram according to the *IFC4.1* specification [bui18a].

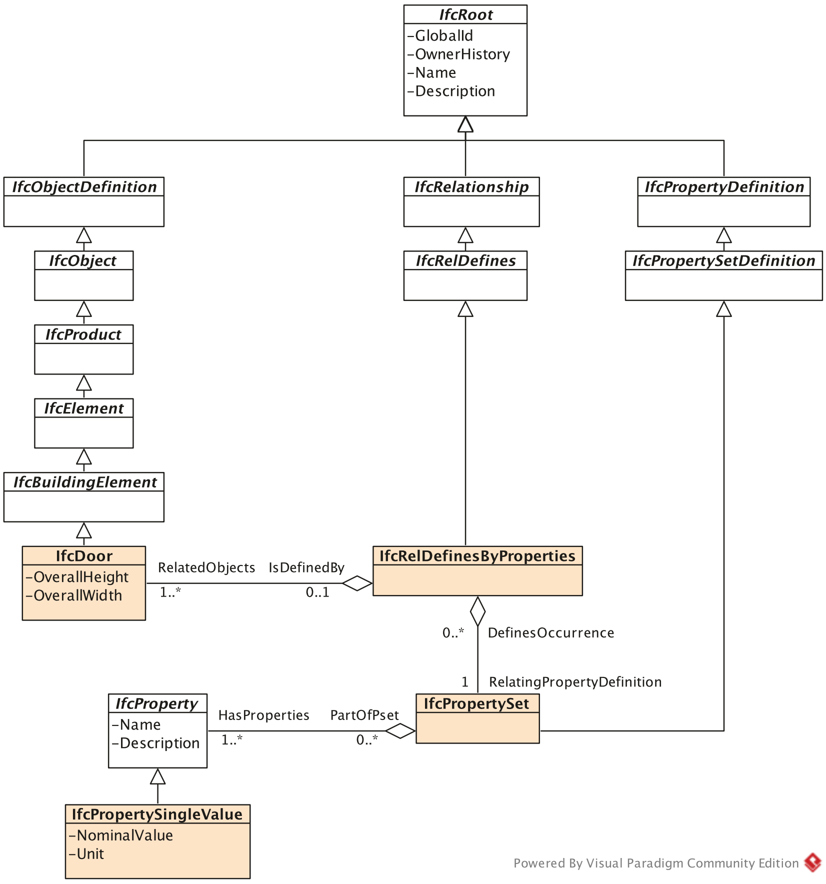


Figure 2.5.: Class hierarchy depicting the `IfcPropertySet` class and its relations according to *IFC4.1* [bui18a]

Furthermore, the usage of IFC as the underlying data model in an application and the creation of an instance of a door are illustrated in figure 2.6; this example applies the mentioned classes which have been highlighted in figure 2.5.

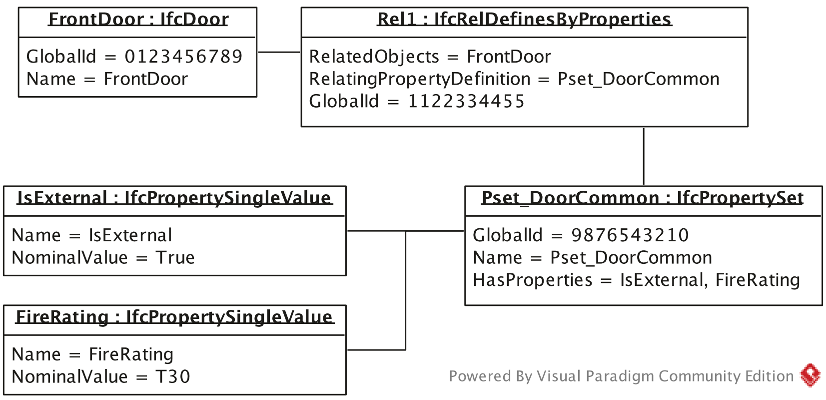


Figure 2.6.: Instance of a simple `IfcDoor` utilizing IFC

Such a naïve implementation would need five objects in total to describe a door with simple properties such as the definition whether the door is external or internal, or its fire rating.

Implementing the same example using OOP would simply need one class `Door` creating the corresponding object as illustrated in figure 2.7.

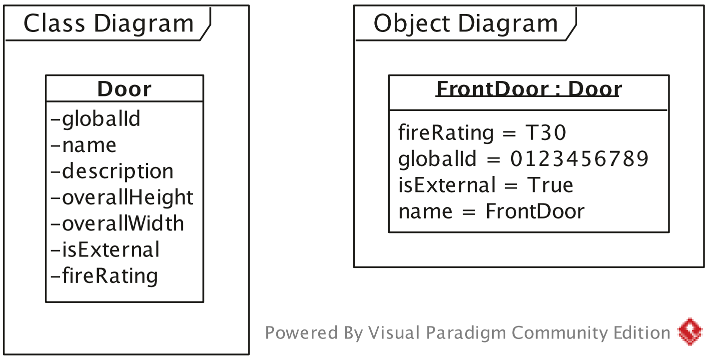


Figure 2.7.: Instance of a simple `Door` in classical OOP

The IFC way of modeling has certain drawbacks. First of all, storing additional properties like that is inconvenient as discussed by Farias, Roxin, and Nicolle, which is why they have simplified this in *IfcWoD* [FRN15]. With respect to the door example that means, checking if a door is an exterior door means iterating over all property sets and all containing properties. At the end, this slows down search queries and the information is not perceivable at a glance [FRN15].

Secondly, the AEC/FM community needs to agree in advance on where to store certain properties and how to name them. However, this is contrary to the idea of BIM and to the fundamental attributes of modeling languages because these agreements must include the correct semantic meaning. Additionally, these kinds of metainformation must be stored as well; otherwise it could get lost sooner or later.

Thirdly, modelers tend to create their own property sets and properties rather than checking and using the predefined ones in the IFC specification. As a result, information can easily get lost or it will get messy if the export of an application does not consider user-defined property sets properly [Hoo18a].

Lastly and with respect to applications such as *ArchiCAD* or *Revit*, property sets are being misused in order to store application-specific information. For example, *ArchiCAD* stores 41 properties in a property set called `ArchiCADProperties`; an excerpt of the stored information is shown in listing 2.12, e.g. the name of the `IfcDoor` entity (`Tür-005`) is stored redundantly in line 4 and 5.

Listing 2.12: Excerpt of the `ArchiCADProperties` in *IFC2x3*

```
1  ...
2  #19390= IFCDOOR('3oJIvZHwmpIvsDNVlm1TkW',#12,'Tür-005',$,'DOOR',#19220,#19386,'F24D2E63-47
       AC-334B-9D8D-5DFBF005DBA0',2.01,0.885);
3  ...
4  #19404= IFCPROPERTYSINGLEVALUE('Element ID',$,IFCLABEL('Tür-005'),$);
5  #19405= IFCPROPERTYSINGLEVALUE('Hotlink und Element-ID',$,IFCLABEL('Tür-005'),$);
6  #19406= IFCPROPERTYSINGLEVALUE('Ebene',$,IFCLABEL('A-Wand Innen Ausbau'),$);
7  #19407= IFCPROPERTYSINGLEVALUE('Bibliothekselement-Name',$,IFCLABEL('Tür Ol 1-Fl 23'),$);
8  ...
9  #19445= IFCPROPERTYSET('3D08DnyK9LUMiKml9Py3ah',#12,'ArchiCADProperties',$
       ,(#19404,#19405,#19406,#19407,[...],#19444));
10 #19447= IFCRELDEFINESBYPROPERTIES('0MRYiZphrxBvRnDzucW5h4',#12,$,$,(#19390),#19445);
11 ...
```

Another example of this misuse is the property set `AC_Pset_Eingang_01_1-Fl_1S_23` that holds 540 (!) properties, e.g. if 2D fixpoints should be displayed in the 3D view. This property set is used to represent the predefined assets of *ArchiCAD*; yet, these properties are unnecessary beyond the scope of this software.

All in all, the discussed issues, workarounds, and misuses diminish the informational substance of the models. Moreover, this situation will be aggravated by the next concept.

2.3.2. Orthogonal classification utilizing object typing

In addition to the property sets, objects in IFC can be further defined by means of object typing; this is illustrated on the right side of figure 2.4. An `IfcObject` can be "typed" by a sub-entity of `IfcTypeObject`, e.g. `IfcDoorType`, which is why the `INVERSE` attribute is called `IsTypedBy`. Additionally, types can have property sets, which are attached directly[6] to the entity compared to an `IfcObject`. These property sets can be static, e.g. `IfcDoorLiningProperties` or dynamically extendable such as the `Pset_DoorCommon`. In general, this enables modelers to add common properties to all objects that are typed by means of a common object type.

The literature review of Götz et al. revealed that "[...] IFC allows typing objects decoupled from the inheritance hierarchy [...]" [Göt+19], which creates an orthogonal classification. As mentioned above, the goal of BIM is a proper collaboration in order to prevent common errors like misconceptions. For example, a stakeholder checking all fire safety regulations will only be interested in all fire doors (a subset of all doors built-in). A classical OOP approach would consider creating an inheritance hierarchy alongside the supertype `Door` to create a subtype `FireDoor`. Furthermore, if a `FireDoor` is glazed, the glazing needs the same fire rating as the door and can be further subtyped as `GlazedFireDoor` (illustrated in figure 2.8).

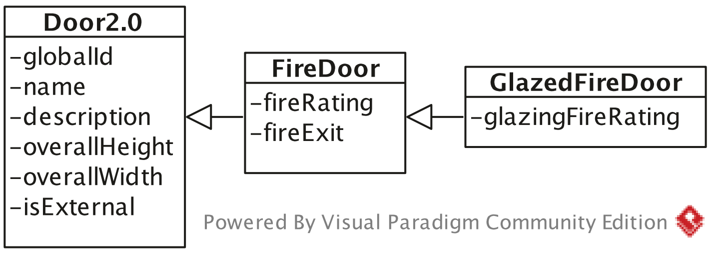


Figure 2.8.: Inheritance hierarchy in classical OOP

However, typing objects in IFC is more comparable to adding properties, which is exemplified in figure 2.9. Applying this construct in a naïve implementation would create instances as illustrated in figure 2.10. Certainly, this example is for demonstrating the idea of object typing only because there is no class `GlazedFireDoor` in IFC; not to mention, there is no possibility to add user-defined `IfcTypeObjects`.

[6]This is for downwards compatibility with respect to previous releases and might be obsolete in the future [bui18a].

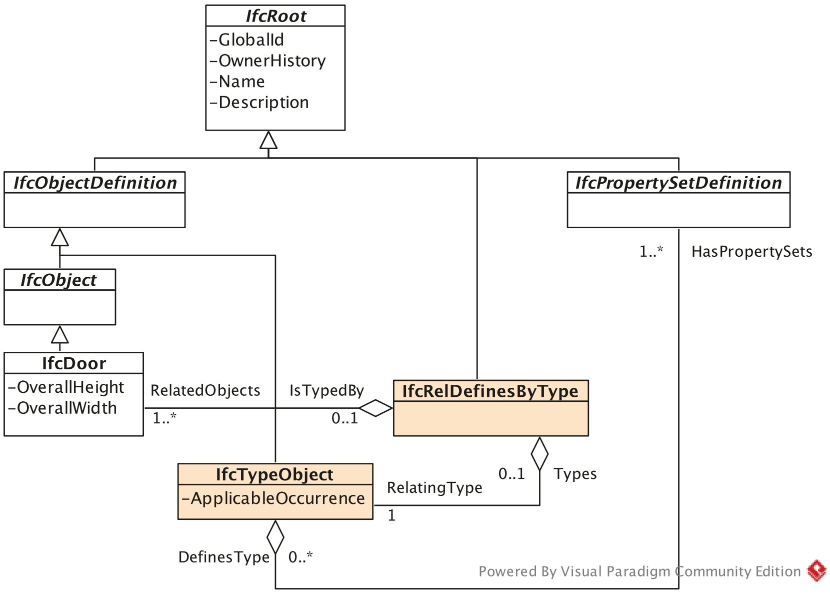


Figure 2.9.: (Simplified) Class hierarchy depicting the `IfcTypeObject` class and its relations according to *IFC4.1* [bui18a]

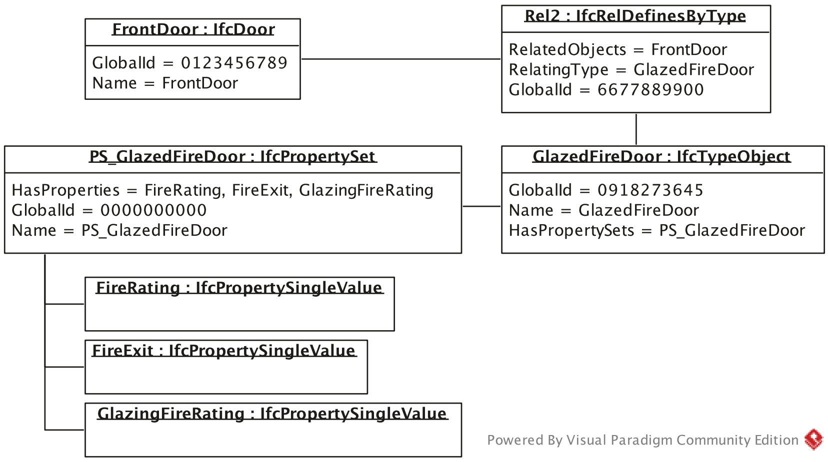


Figure 2.10.: Instance typed by an `IfcTypeObject` utilizing IFC

Again, this modeling concept is not free from defects. Firstly, the way properties are handled with respect to the "same" property set attached to an `IfcObject` and an `IfcTypeObject` simultaneously needs special attention. Table 2.1 illustrates this example accordingly, which can be classified as a simple version of the so-called "Prototype pattern" by Gamma et al. [Gam+94]. That means, the effective properties are determined dependent on the attached property sets and the property sets of the attached type. More precisely, if the same set is attached to both the occurrence and the type, and if they share the same property, then the effective property will be taken from the occurrence. As a result, applications utilizing IFC as the underlying data model must take care of this way of property handling.

Table 2.1.: Example for illustrating property assignment (taken and adapted from [bui18a])

Properties assigned to IfcDoor	Properties assigned to IfcDoorType	Resulting property value for individual door
Pset_DoorCommon	Pset_DoorCommon	
- *FireExit* = TRUE		TRUE
	- *FireRating* = T30	T30
- *IsExternal* = TRUE	- *IsExternal* = FALSE	TRUE

Secondly, an IfcTypeObject can be classified as a "powertype" according to the definition by Atkinson and Kühne, that means an instance will be typed as an IfcDoor but is semantically – and should correctly – be typed as a GlazedFireDoor [AK01]. Götz et al. have identified this powertype pattern as "a restricted version of multi-level modeling" [Göt+19]. As mentioned in the introduction, removing the restrictions and implementing a proper multi-level modeling approach could also improve extensibility and maintainability of IFC. The reason for that can be attributed to a shift of the object typing concept to a more appropriate level in the metadata architecture. Currently, object typing takes place at level M1, which is "incorrect"; this concept should be handled as a metamodeling concept instead. Therefore, the definition of this concept should be located at level M2 rather than M1. However, this approach will be excluded in this thesis and left open for future research.

Lastly, the semantic meaning of a "type" becomes blurred due to an excessive usage of different type attributes. In consequence, the modelers might misuse or ignore these attributes as a result of a missing clarification and a missing sound separation or good explanation. The following list will give an overview of the different "type" attributes – with no claim for completeness:

- ObjectType attribute of IfcObject: defined as "[...] a particular type that indicates the object further [...]" [bui18a]
- IsTypedBy (INVERSE) attribute of IfcObject: main attribute for typing objects
- PredefinedType attribute of sub-entities of IfcObject (e.g. IfcDoor): an enumeration holding common subtypes, e.g. GATE or DOOR correspondingly
- ElementType attribute of IfcElementType (e.g. supertype of IfcDoorType): also, defined as "[...] a particular type that indicates the object further [...]" [bui18a]; used to type object types [Hoo18b]

ObjectType and ElementType are intended to be used for the addition of user-defined information. This is the case if the PredefinedType will be set to USERDEFINED because the given definitions were not suitable [bui18a]. However, *IFC2x3 TC1*, for example, holds 465 element based PredefinedTypes [Hoo18b]. At the end, these predefined types might be ignored by modelers that add their own types consequently, which diminishes the informational value of the model. Additionally, exporting and importing types is not trivial in applications such as *Revit* [Hoo18b; Mou19].

Altogether, typing objects in IFC is not self-explaining and more complicated than it should be. However, with respect to BIM, a more sophisticated approach is crucial in order to improve the quality of the information models.

In conclusion, the core issues of IFC identified and discussed in this thesis are the handling of **property sets** and the given **object typing** concept. Especially the typing of objects in its current state should be considered critical because semantic issues hamper the value of the models. However, these models are expected to have longevity, which is why expressivity, clarity, and parsimony are essential attributes. Therefore, a mature object-oriented approach is needed in order to address the found deficiencies.

Regarding RQ1, the core issues have been extensively analyzed in this section; yet, the question how these issues can be improved in order to achieve the goal of this thesis is still to be answered. A core problem is that IFC and EXPRESS show a lack of flexibility and dynamicity to some extent. Consequently, this leads to the assumption that a modern object-oriented approach like role-oriented modeling could potentially improve certain aspects of IFC. As a result, a proof of concept called Industry Foundation Classes with Roles (IFC-R) has been developed in order to verify the aforementioned assumption.

3. Developing a Role-oriented Solution

After the basics with special focus on the issues of IFC have been covered, this chapter will resolve these with the aid of roles. Hence, section 3.1 will introduce the proof of concept of a role-oriented approach, namely Industry Foundation Classes with Roles (IFC-R). This includes the origin of the idea, the necessary basics regarding role-oriented modeling, and an in-depth comment on the developed models and tools. Thereafter, an explanation of the implemented prototypes using IFC and IFC-R as well as a small documentation on how to use them will be subject of section 3.2. This will then enable a comprehensive evaluation of IFC-R compared to IFC.

3.1. Industry Foundation Classes with Roles (IFC-R)

IFC – more precisely EXPRESS – is defined as an information modeling language. That means modelers can employ the concepts and predefined schemes of IFC in order to create information rich data models in the building industry. However, IFC is no programming language and the standard does not define any "specific way of implementation" [LK12], which categorizes it is an "implementation-independent data model" [LK12]. As a result, a variety of ways for handling the data models exists in the BIM application landscape. A first analysis of an actual IFC model will emphasize the associated problems.

First of all, it is necessary to comprehend the general structure of the data models – more precisely, to analyze the stored data and information in a STEP file. Therefore, a model of a simple house has been created with *ArchiCAD 23* (developed by *Graphisoft*) for this task. Figure 3.1 illustrates an excerpt of the 3D geometric representation of this example that has been used throughout the development process. This simple house consists of five rooms – hallway, living room, bathroom, bedroom, and kitchen. Additionally, some furniture, such as a bed and a dining table, has been added. Last but not least, the house has a roof, several windows, and doors, of course.

Figure 3.1.: Simple house created with *ArchiCAD 23* (own screenshot)

Such a model is generally stored in a **.pln*-file, which is a project file containing all necessary information for *ArchiCAD*. Nevertheless, other file formats are needed outside the scope of this application, which is why *ArchiCAD* offers several export formats, e.g. IFC. Selecting IFC as the format for exporting reveals a list of different translators as seen in figure 3.2.

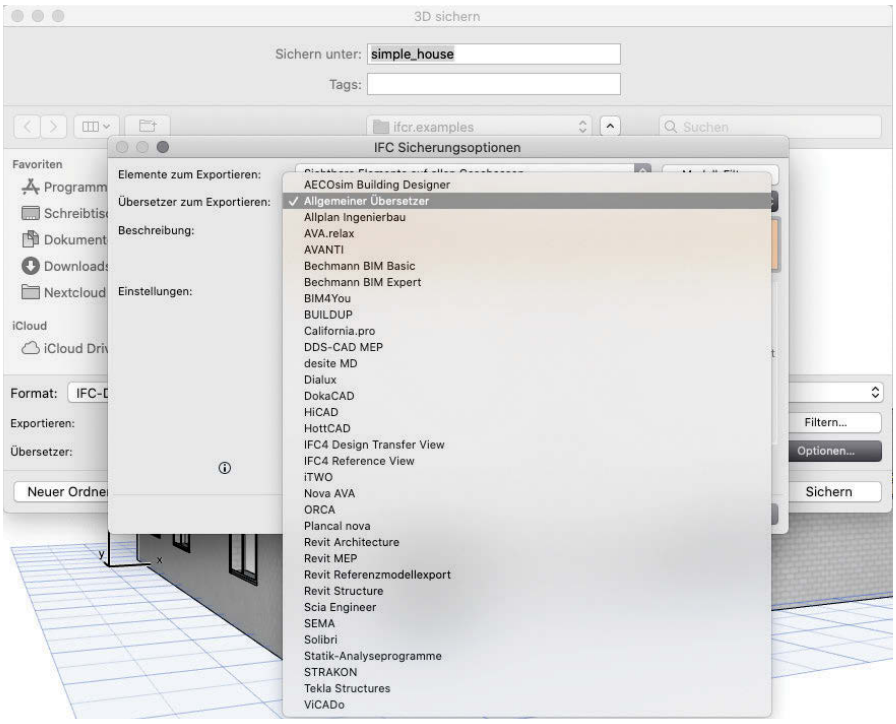


Figure 3.2.: List of IFC translators offered by *ArchiCAD 23* (own screenshot)

In order to highlight the aforementioned issues, the same project has been exported using three different translators: (1) the general translator (*Allgemeiner Übersetzer*), (2) the *IFC4 Design Transfer View* translator, and (3) the *Revit Architecture* translator. The general translator and the *Revit Architecture* translator export the project with the file schema version *IFC2X3*; the *IFC4 Design Transfer View* translator uses the file schema version *IFC4*. The doors of the house are exported with similar attributes as seen in listing 3.1:

Listing 3.1: Excerpt of a door in all exports

```
// General Translator
#2601= IFCDOOR(’0p88roaDKKG8EzXCthnb8k’,#12,’Tür-001’,$,’DOOR’,#2347,#2591,’33208D72-90D5-1440-83BD-84CDEBC6522E’,2.135,1.51);

// IFC4 Design Transfer View
#813= IFCDOOR(’0p88roaDKKG8EzXCthnb8k’,#12,’Tür-001’,$,$,#656,#804,’33208D72-90D5-1440-83BD-84CDEBC6522E’,2.135,1.51,.DOOR.,$,$);

// Revit Architecture
#2199= IFCDOOR(’0p88roaDKKG8EzXCthnb8k’,#12,’Tür-001’,$,’DOOR’,#1951,#2190,’33208D72-90D5-1440-83BD-84CDEBC6522E’,2.135,1.51);
```

Even though some minor variations in attribute values are not crucial, an examination of the walls and the roof of the house reveals issues with respect to interoperability as seen in listing 3.2 and 3.3:

Listing 3.2: Excerpt of a wall in all exports

```
// General Translator
#3800= IFCWALL(’008OzmUJ1MIBTa2twwMq8c’,#12,’Wand-002’,$,$,#3593,#3789,’00218F70-7930-5648-B764-0B7EBA5B4226’);

// IFC4 Design Transfer View
#895= IFCWALL(’008OzmUJ1MIBTa2twwMq8c’,#12,’Wand-002’,$,$,#857,#891,’00218F70-7930-5648-B764-0B7EBA5B4226’,.NOTDEFINED.);

// Revit Architecture
#3217= IFCWALLSTANDARDCASE(’008OzmUJ1MIBTa2twwMq8c’,#12,’Wand-002’,$,$,#3179,#3213,’00218F70-7930-5648-B764-0B7EBA5B4226’);
```

Listing 3.3: Excerpt of the roof in all exports

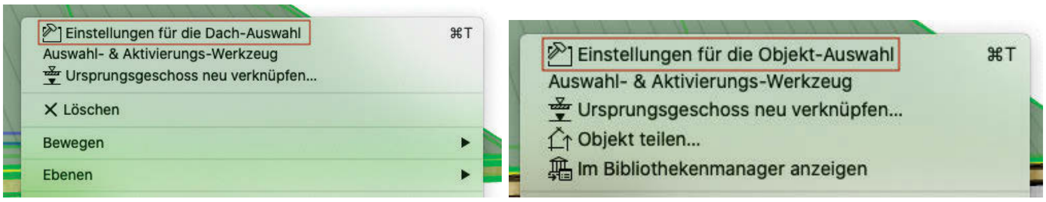
```
// General Translator
#41812= IFCSLAB(’0jLAtUefiYIwd90bi$XrfU’,#12,’Dach-001’,$,$,#41811,$,’2D54ADDE-A29B-224B-A9C9-025B3F875A5E’,.ROOF.);

// IFC4 Design Transfer View
#10618= IFCROOF(’0jLAtUefiYIwd90bi$XrfU’,#12,’Dach-001’,$,$,#10190,#10579,’2D54ADDE-A29B-224B-A9C9-025B3F875A5E’,.NOTDEFINED.);

// Revit Architecture
#36319= IFCSLAB(’0jLAtUefiYIwd90bi$XrfU’,#12,’Dach-001’,$,$,#35557,#36275,’2D54ADDE-A29B-224B-A9C9-025B3F875A5E’,.ROOF.);
```

The general translator and the *IFC4 Design Transfer View* translator both export a wall by using the entity `IfcWall`. In contrast, the translator for *Revit* uses the `IfcWallStandardCase` entity. The same issue applies to the export of the roof – two translators use the `IfcSlab` entity, the other one uses `IfcRoof`, despite the fact that every entity schema used by the different translators exists in both versions of IFC. The difference between the exports of walls might be reasonable; yet, the different entities for the roof are crucial because a slab is not necessarily the same as a roof.

Moreover, importing such an **.ifc*-file in *ArchiCAD*, e.g. which has been created using the general translator, is different compared to loading the default **.pln*-file. Although the IFC export contains several *ArchiCAD*-specific properties, the application does not recognize the entities "correctly". Right-clicking the roof in an *ArchiCAD* project offers the context menu option for configuring the object using the "roof-selection-tool" (figure 3.3a), whereas opening the same project with a STEP file leads to a different behavior. A right-click would only offer configuring the object using a general "object-selection-tool" (figure 3.3b). Figure 3.3 illustrates the difference in the classification of objects in *ArchiCAD 23*.



(a) "roof-selection-tool" in **.pln* (b) "object-selection-tool" in **.ifc*

Figure 3.3.: Different selections/configurations of objects recognized by *ArchiCAD 23* (own screenshots)

To put it in a nutshell, IFC is the *de facto* standard for interoperability in the building industry and with respect to BIM. However, the standard itself does not define any guidelines on how to implement and handle the data models. As a result, many applications handle objects, types, and properties differently, which justifies the need for a more sophisticated approach to the classification. For example, a slab can be classified as a roof in a certain context, e.g. a roof slab. Therefore, interpreting objects with respect to a specific context and separating concerns is essential to improve the found issues. Naturally, role-oriented modeling will be considered as a way for improvements, which will be explained in the next section.

3.1.1. Role-oriented Modeling with CROM

Before diving into the essential basics of role-oriented modeling, a short recall of EXPRESS is necessary. As mentioned in section 2.1.1, EXPRESS is standardized as *ISO 10303-11* [ISO04], which defines some fundamental principles of this language. The first principle or "concept" focuses on schemes defined in EXPRESS and is formulated as follows:

> *A schema written in the EXPRESS language describes a set of conditions which establishes a domain. Instances can be evaluated to determine if they are in the domain. If the instances meet all the conditions, then they are asserted to be in the domain. If the instances fail to meet any of the conditions, then the instances have violated the conditions and thus are not in the domain.* [...] [ISO04, p. 7]

The core of this principle says that each entity will be evaluated against certain domains – if it matches, it will be associated to that domain; otherwise it will not. That means, entities defined in a particular schema have a specific **context**. However, EXPRESS' building blocks – as explained in section 2.2 – are not sufficient to make fine-grained distinctions with respect to several contexts, in my opinion. For example, the definition of what properties are essential for walls differs from the viewpoint of different stakeholders. Therefore, **roles** have been considered as a proper approach for making these distinctions because they enable viewing objects dynamically in different contexts.

The main idea is to represent properties and types by means of roles because both concepts further define or classify an object and preset the context; additional user-defined roles beyond the scope of IFC are possible as well. Furthermore, the corresponding objectified relationships will be replaced by a so-called "compartment"; yet, basics of role-oriented modeling need clarification before diving deeper into the explanation of IFC-R.

IFC and EXPRESS have been denoted as object-oriented or "object-flavored". That means these modeling languages are able to describe "things" – generally called **objects** – from the real world. The concept of object orientation has its origin way back in the 1960s, and one of the first object-oriented programming languages was *Smalltalk* by Alan Kay. Since then, OOP has evolved to being one of the most used programming paradigms, which has yield to many object-oriented programming languages such as its most prominent representative: *Java*. However, the core concept of OOP, **classes**, are not sufficient enough to represent context-dependent and collaborative behavior of objects [RWL+96; RG98; Küh+15].

A class acts like a blueprint, which means it defines which properties and methods (describe behavior) each instance of an object can have. An example of a class and an instantiated object has been given in figure 2.7 in section 2.3.1. This specification of objects usually suffices many applications and use cases; yet, the demand for context-dependent properties and behavior is increasing [Küh+15]. As mentioned above, context-dependency is also essential in the area of the building industry as many different stakeholders are involved in the lifecycle of a building. Consequently, each stakeholder will work at the building with a different context such as electricity or plumbing. Those contexts, in the end, can be captured by the usage of **roles**.

Roles and Role Types Roles are usually known from movies and theater plays. An actress or an actor plays a role in a movie. However, after a day of shooting, they all stop playing that role and switch back to their normal lives. A similar concept can refer to the daily life of "normal" people as well. For example, a person will take on the role of an employee at work. After work, this person drops that role and, perhaps, switches to the role of a customer because she or he goes to the grocery store to buy food for dinner. Each role is involved in a different context probably needing adjusted properties or behavior, e.g. an employee usually has a staff number, which will not be needed after work.

This basic explanation of roles should introduce the main idea behind role-oriented modeling, which is a very familiar concept in the research area of modeling languages [Küh+15]. Unfortunately, the **nature of roles** is not fully utilized by many of them. Kühn et al. identified to following natures of roles [Küh+15]:

- **relational nature**: the ends of relationships such as in UML; represents relations between players and roles
- **context-dependent nature**: the ability to capture context-dependent behavior of objects
- **behavioral nature**: the ability to adapt the behavior of the playing objects

Additionally, roles are able to separate concerns [RWL+96; RG98], which describes the ability to differentiate the purposes of object collaborations. In sum, the nature of roles allows objects to adopt different context-dependent behavior and to differentiate between several object collaborations dynamically. Such a dynamic view on an object is denoted as **role type** by Riehle and Gross [RG98]. All these characteristics have been transferred by Thomas Kühn and his team into a formal role-based modeling language called **Compartment Role Object Models (CROM)** [Küh+15].

CROM and CROI Kühn et al. introduced the metamodel CROM together with the Compartment Role Object Instances (CROI) in [Küh+15]. The former is for describing formal models on the level of role types, whereas the latter is for modeling instances. Furthermore, CROM allows the constraining of role models, offers a formal validation, and it has a graphical notation for better comprehension [Küh+15; Küh+16; Küh+19]. The ontological foundation distinguishes the following type concepts:

- **Natural Type**: a natural type can be considered as a typical object in OOP, e.g. a `Person`
- **Role Type**: role types, as defined by Riehle and Gross, are a dynamic view on objects such as the role type `Employee` (work context) or `Customer` (after work context)
- **Compartment Type**: a compartment captures the context-dependent collaboration of natural and role types, e.g. a `Company` would be a suitable compartment for illustrating the above-mentioned work context
- **Relationship Type**: relationship types help model the relational nature of roles; e.g. the context-dependent relationship between an `Employee` who is serving a `Customer`, which can be denoted as `serves`

Figure 3.4 depicts a CROM in its graphical notation using the `Employee-Customer-Company` example; the model has been created using the web-based implementation of FRaMED (Full-fledged Role Modeling EDitor) [Küh+16; Küh20]. Angular boxes (gray) represent natural types, yellow boxes represent compartment types, and rounded boxes represent role types. Additionally, the arrows represent so-called "fills" relationships, which means a natural type *plays* a role type. The line between the role types represents a relationship type.

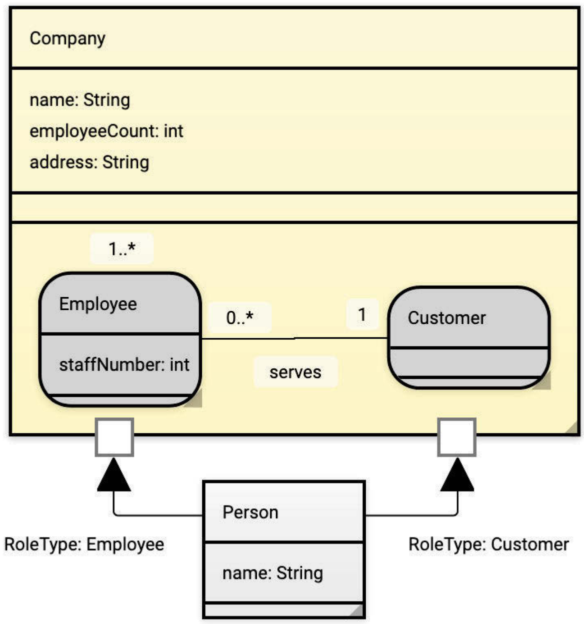


Figure 3.4.: Example for a CROM created with FRaMED-io [Küh20]

Furthermore, CROM offers the possibility to constrain roles, which is denoted as **role constraints**. The following constraints exist:

- **role-implication**: an object that plays role *A* must also play role *B*
- **role-equivalence**: an object that plays role *A* must also play role *B* **and** vice versa
- **role-prohibition**: an object that plays role *A* must **not** play role *B* and vice versa

Moreover, CROM modelers can use inter- and intra-relationship constraints such as denoting a relationship as irreflexive, and models can be constrained with respect to the cardinality of types, e.g. a `Company` has at least one `Employee`.

Last but not least, the ontological foundation differentiates between three important properties: **rigidity**, **foundedness**, and **identity** [Küh+15]. All types besides role types are rigid, which means that instances have this type until the end of their lifetime, e.g. a `Person` will never stop being a person. A role type, however, only exists until a natural type stops playing it. Foundedness describes the property of an instance as dependent on the existence of another instance – similar to the `INVERSE` attribute of EXPRESS, which applies to all types besides natural types. For example, a `Company` only exists if it has employees and customers. The last property, namely the identity, can be unique, derived or composed. Natural and compartment types have a unique identity, whereas the identity of a role derives from its natural or its compartment. A composite identity applies to relationship types and means that the identity of a relationship relies on its both ends.

In addition to CROM, Kühn et al. have developed CROI for representing the instantiation of the models. On this level of instances, the model distinguishes between **naturals**, **roles**, **compartments** and **links** as instances of their corresponding types [Küh+15]. A brief summary of the building blocks of CROM is given in table 3.1.

Table 3.1.: Summary of CROM's building blocks

Type Level (Instance)	Graphical Notation	Ontological Properties	Example
Natural Type (Natural)	gray, angular box	rigid non-founded unique identity	`Person`
Role Type (Role)	gray, rounded box	anti-rigid founded derived identity	`Employee`
Compartment Type (Compartment)	yellow, angular box	rigid founded unique identity	`Company`
Relationship Type (Link)	simple line	rigid founded composed identity	`serves`

As a profound understanding of IFC-R and role-oriented modeling is now set as a basis, the upcoming section will comment the models and tools that have been developed in order to realize the proof of concept.

3.1.2. IFC-R: Models and Tools

As mentioned in chapter 1, changing IFC from scratch or having a big impact on the BIM application landscape with big players like *ArchiCAD* and *Revit* exceeds the possibilities of this thesis. Therefore, this proof of concept considers subsequent steps in the lifecycle of a building; more precisely, the main goal was the development of a proper runtime model having an existing IFC model as the basis, e.g. for a facility management application. The result of this is a workflow of models, scripts, and model-transformations, which has been subsumed under the name Industry Foundation Classes with Roles (IFC-R). An overview of the steps of this workflow is given in figure 3.5.

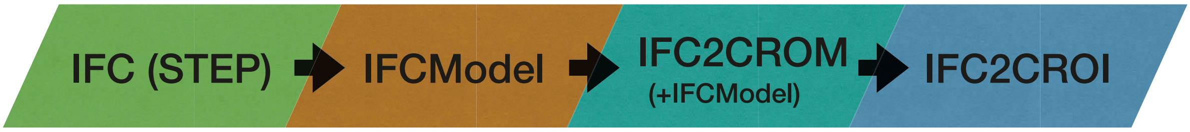


Figure 3.5.: Overview of the steps in the IFC-R workflow

Due to the fact that IFC-R relies on an existing IFC model, it can be considered an additional layer or a decorator for applications, which enables improvements in terms of extensibility and maintainability. The following use cases mainly influenced the development of IFC-R:

- facility management: an application needs a proper (runtime) model because daily changes of the model are likely
- expense budgeting in projects: editing and reading vendor-specific or cost-related information is disproportional with the huge amount of properties and property sets stored actually in some models

The first use case is exemplary for the usage of roles due to the context-dependency of the different tasks in facility management. Additionally, the model needs to be highly dynamic with respect to the daily changes, e.g. annotating the maintenance status of furniture. Moreover, filtering for specific properties with respect to a certain context is also helpful regarding the second use case. These use cases will be handled utilizing IFC-R, i.e. an application will load and store all information in a CROM (for metadata) and a CROI. Hence, it is necessary to explain the workflow of how to create the models by employing an IFC model as the origin.

IFCModel

First of all, each step in the workflow is a model-to-model (M2M) transformation. Therefore, an existing IFC model serves as the starting point; as mentioned above, the simple house example will fulfill this task. The goal model is a CROM respectively a CROI, which has been provided by Thomas Kühn as an *Ecore* metamodel [Küh19]. The *Ecore* metamodel is part of the *Eclipse Modeling Framework (EMF)*, and most of the parts of the workflow have been developed with the EMF.

To start the workflow properly, an *Ecore* metamodel of IFC is necessary, hence, this was created as a first step. The *Ecore* model contains – as a first draft for this thesis – only the essential parts of IFC, e.g. `IfcObject`, `IfcPropertySet`, and `IfcTypeObject`. An excerpt of the so called "IFCModel" metamodel is shown in figure 3.6, and a complete class diagram can be found in figure A.1 (in the appendix).

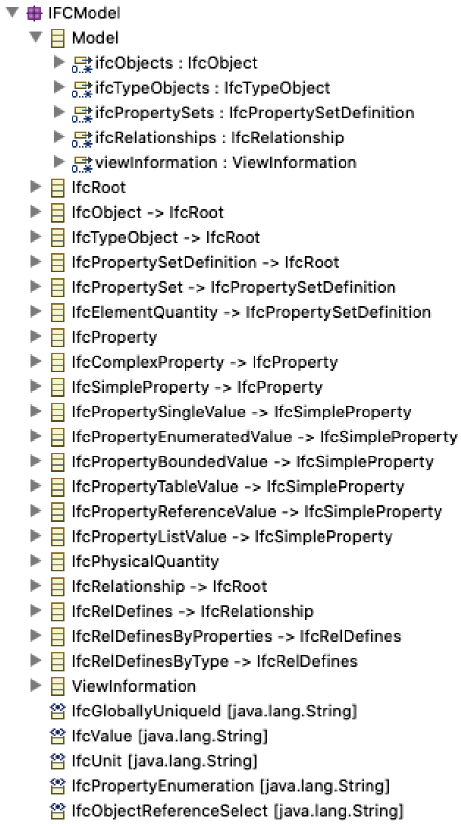


Figure 3.6.: Excerpt of the IFCModel (*Ecore* metamodel)

Creating a specific model with the help of this metamodel and an IFC model will be achieved by a *Python* script utilizing the library *IfcOpenShell-python*[1] for parsing STEP files and the framework *PyEcore*[2] for creating model instances out of *Ecore* metamodels. The corresponding script can be found in the appendix (listing B.1) and handles the following substeps in order to create an instance of an IFCModel (stored as an XMI file):

1. Startup step
   1.1. Read in the STEP file (input)
   1.2. Initialize the *Ecore* metamodel (`IFCModel.ecore`)
   1.3. Apply the domain view configuration (will be explained later)
   1.4. Parse the IFC model in order to collect all `IfcObject` entities
2. Create a `Model` instance (holds all `IfcObjects`, `IfcTypeObjects`, `IfcPropertySets`, `IfcRelationships`, and the `viewInformation`)
3. Iterate over all `IfcObjects` of the STEP file
   3.1. Add general properties like the `GlobalId` to the modeled instance of the object
   3.2. Iterate over its relationships (`IsDefinedBy`)
      3.2.1. Add the relationship itself to the `Model` instance
      3.2.2. Handle and add `IfcPropertySets`; add to the `Model` instance
      3.2.3. Handle and add `IfcTypeObjects`; add to the `Model` instance
   3.3. Add the `IfcObject` to the `Model` instance

[1] *IfcOpenShell-python* has been used in the version 0.6.0 [Kri20].
[2] *PyEcore* has been used in the version 0.11.7 [Ara20].

4. Add the domain view information if given
5. Store the created model instance in an `*.ifcmodel` file

An excerpt of an actual IFCModel of the simple house example can be found in listing 3.4. This model is only an intermediate result of the overall workflow and serves as an input for the M2M transformation to create a corresponding CROM – this next transformation step is denoted as "IFC2CROM".

Listing 3.4: Excerpt of an IFCModel of the simple house example (`simple_house.ifcmodel`)

```xml
<?xml version='1.0' encoding='UTF-8'?>
<ifcmodel:Model xmlns:ifcmodel="ifcr.metamodels.ifcmodel">
  ...
  <ifcPropertySets xsi:type="ifcmodel:IfcElementQuantity" globalId="1kQMlmT0rD35a9E43iKTas"
     methodOfMeasurement="ARCHICAD BIM Base Quantities" name="BaseQuantities">
    <quantities name="GrossFloorArea"/>
  </ifcPropertySets>
  <ifcPropertySets xsi:type="ifcmodel:IfcPropertySet" name="Pset_ConcreteElementGeneral"
    globalId="0FFu7SLgVp5GfpV9sNuDHG">
    <hasProperties xsi:type="ifcmodel:IfcPropertySingleValue" name="FireRating"
    nominalValue="REI120"/>
  </ifcPropertySets>
  ...
  <ifcObjects name="Projekt" subtype="IfcProject" globalId="344O7vICcwH8qAEnwJDjSU"/>
  <ifcObjects name="Gelände" subtype="IfcSite" globalId="20FpTZCqJy2vhVJYtjuIce"
    isDefinedBy="2Hm9JvZjohDNSD2kdxZI3b"/>
  <ifcObjects name="Gebäude" subtype="IfcBuilding" globalId="00tMo7QcxqWdIGvc4sMN2A"
    isDefinedBy="0L87OdSD3DqSTjSRlAciZL"/>
  ...
  <ifcRelationships xsi:type="ifcmodel:IfcRelDefinesByProperties" globalId="2
    Hm9JvZjohDNSD2kdxZI3b" relatingPropertyDefinition="2GNZepdf73fvGc$0W6rozj"
    relatedObjects="20FpTZCqJy2vhVJYtjuIce"/>
  <ifcRelationships xsi:type="ifcmodel:IfcRelDefinesByProperties" globalId="0
    L87OdSD3DqSTjSRlAciZL" relatingPropertyDefinition="1kQMlmT0rD35a9E43iKTas"
    relatedObjects="00tMo7QcxqWdIGvc4sMN2A"/>
  <ifcRelationships xsi:type="ifcmodel:IfcRelDefinesByProperties" globalId="1
    rsfW7t52Oyc_Lh8aBxAXJ" relatingPropertyDefinition="1qq9OUkQygW2qIwWNUGst_"
    relatedObjects="2Wk49hkHPC2hvWchm5NASJ"/>
  ...
  <ifcTypeObjects name="Kalksandstein 240" globalId="0MbADYmtOfD7Rbh9Tz5CXe" subtype="
    IfcWallType"/>
  <ifcTypeObjects name="2-Flügelfenster 1+1 23" globalId="1_0xmR7il3lcZdaNmo6fzL" subtype="
    IfcWindowStyle"/>
  <ifcTypeObjects name="Eingang 01 1-Fl 1S 23" globalId="1fUq2gX$TzJRjHTccJAOHV" subtype="
    IfcDoorStyle"/>
  ...
</ifcmodel:Model>
```

IFC2CROM

The next step in the IFC-R workflow utilizes the previously created IFCModel, which is stored in a corresponding `*.ifcmodel` file. Additionally, the metamodels IFCModel (`IFCModel.ecore`) and CROM (`CROM.ecore`) are needed as well in order to fulfill the next step: a transformation called IFC2CROM.

This M2M transformation is written with a *Java*-based scripting language called *Epsilon*[3] (by *Eclipse*). This language works out of the box with the EMF and offers a variety of task-specific languages, for example the Epsilon Transformation Language (ETL), which has been used for both IFC2CROM and IFC2CROI [Ecl20]. ETL can be used to transform any number of input models to different output models [Ecl20]. The core of such transformations is defined using a rule-based approach, which means rules define which elements of the source model will be transformed to specific elements of the target model. For example, listing 3.5 shows an exemplified rule for transforming an `IfcObject` of the IFCModel to a `NaturalType` of the CROM. For example, line 4 sets the name property of the created `NaturalType` to the corresponding subtype of the `IfcObject`, e.g. `IfcDoor`.

Listing 3.5: Rule to transform `IfcObjects`

```
1 rule IfcObject2NaturalType
2 transform ifcObject : IFCModel!IfcObject
3 to naturalType : CROM!NaturalType {
4     naturalType.name = ifcObject.subtype
5 }
```

The ETL script for IFC2CROM can be found in the appendix (listing B.2) and consists of the following rules and operations (helper functions):

Rules:

- `IfcObjectModel2IfcObjectNT`: transforms an `IfcObject` to a Natural Type
- `IfcTypeObjectModel2IfcTypeObjectRT`: transforms an `IfcTypeObject` to a Role Type
- `IfcPropertySetModel2IfcPropertySetRT`: transforms an `IfcPropertySet` to a Role Type
- `IfcElementQuantityModel2IfcElementQuantityRT`: transforms an `IfcElementQuantity` to a Role Type
- `ViewInformation2ViewInformationDT`: transforms the `viewInformation` to a Data Type

Operations:

- `addIfcRootAttributes`: adds the attributes inherited by `IfcRoot` to the calling element, e.g. `GlobalId`
- `addDefaultDataTypes`: defines and adds a set of default Data Types to the CROM `Model`, e.g. `IfcProperty`
- `addDefaultNaturalTypes`: defines and adds a set of default Natural Types to the CROM `Model`, e.g. `IfcObject`
- `addDefaultCompartmentTypes`: defines and adds a set of default Compartment Types to the CROM `Model`, e.g. `ObjectDefinition`
- `addDefaultRoles`: defines and adds a set of default Role Types to a CROM `CompartmentType`, e.g. `IfcTypeObject`
- `retrieveDefaultType`: retrieves the default type for a CROM element by name
- `retrieveSpecificationName`: retrieves the specification name for an element because the name is often an optional attribute in IFC and an additional differentiation between `IfcTypeObject`, `IfcPropertySet` and `IfcElementQuantity` by means of prefixes is needed

[3] *Epsilon* has been used in the version 1.5.1 [Ecl20]. It has been chosen over other languages like *ATL* due to an easy syntax, an adequate scope, and its straightforward integration of additional *Java* code.

Applying these rules and operations on an IFC model results in a CROM metamodel, which contains various information, e.g. which `IfcPropertySet`-roles can be played by an `IfcObject`. A schematic depiction of such a CROM can be found in figure 3.7.

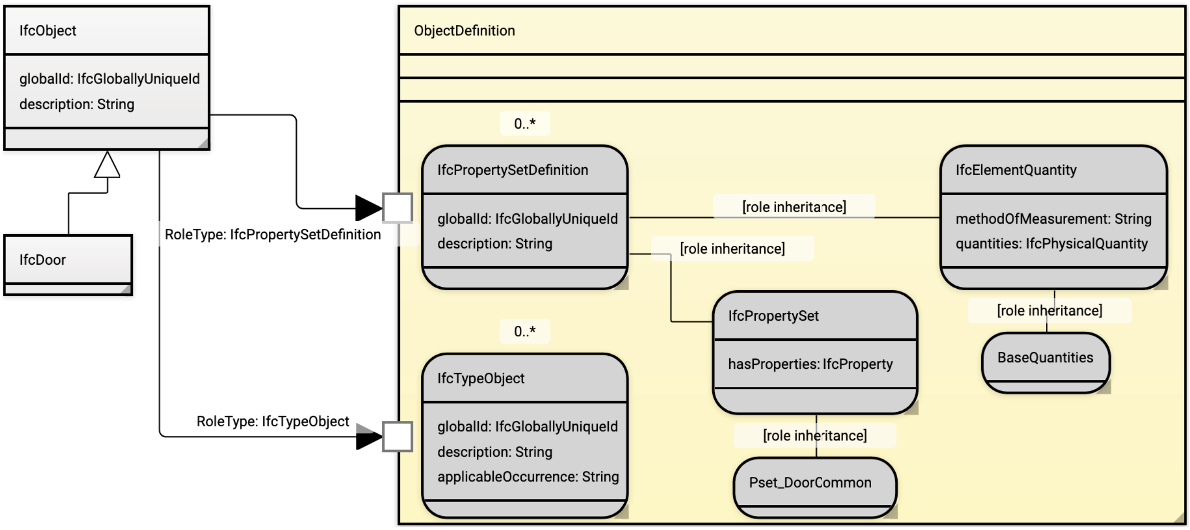


Figure 3.7.: Schematic depiction of an IFC model as CROM

The output of the IFC2CROM transformation is a `*.crom` file – again stored as a simple XMI file; an excerpt of a CROM of the simple house example can be found in listing 3.6.

Listing 3.6: Excerpt of a CROM of the simple house example (`simple_house.crom`)

```
<?xml version="1.0" encoding="ASCII"?>
<crom:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" xmlns:crom="ifcr.metamodels.crom">
  ...
  <elements xsi:type="crom:CompartmentType" name="ObjectDefinition">
    <parts>
      <role xsi:type="crom:RoleType" name="IfcTypeObject">
        <attributes name="globalId" type="//@elements.0"/>
        <attributes name="name"/>
        <attributes name="description"/>
        <attributes name="applicableOccurrence"/>
      </role>
    </parts>
    <parts>
      <role xsi:type="crom:RoleType" name="IfcPropertySetDefinition">
        <attributes name="globalId" type="//@elements.0"/>
        <attributes name="name"/>
        <attributes name="description"/>
      </role>
    </parts>
    <parts>
      <role xsi:type="crom:RoleType" name="IfcPropertySet">
        <attributes name="hasProperties" type="//@elements.1"/>
      </role>
    </parts>
    <parts>
      <role xsi:type="crom:RoleType" name="IfcElementQuantity">
        <attributes name="methodOfMeasurement"/>
```

```
        <attributes name="quantities" type="//@elements.9"/>
      </role>
    </parts>
    ...
    <parts>
      <role xsi:type="crom:RoleType" name="T_EingangFlS"/>
    </parts>
    ...
    <parts>
      <role xsi:type="crom:RoleType" name="PS_PsetDoorCommon"/>
    </parts>
    ...
    <parts>
      <role xsi:type="crom:RoleType" name="EQ_BaseQuantities"/>
    </parts>
    ...
  </elements>
  ...
  <elements xsi:type="crom:NaturalType" name="IfcDoor"/>
  ...
  <relations xsi:type="crom:Fulfillment" filled="//@elements.11/@parts.0/@role" filler="//
    @elements.10"/>
  <relations xsi:type="crom:Fulfillment" filled="//@elements.11/@parts.1/@role" filler="//
    @elements.10"/>
  ...
</crom:Model>
```

The output of this workflow step does not contain any specific information of the IFC model such as the height of the front door of the simple house example. To be precise, the actual values of the model will not be covered by the CROM because the assignment of values happens during instantiation. Therefore, a CROI is needed as well, which is why the next step in the workflow focuses on the IFC2CROI transformation.

IFC2CROI

The last step in the IFC-R workflow is another M2M transformation. In order to create a `*.croi` file, this transformation step takes the IFCModel from step one and the CROM (as metamodel) from the previous step as input for the creation of a CROI, which is why this transformation is called "IFC2CROI".

Again, this transformation step has been written using ETL and the final script can be found in the appendix (listing B.3). IFC2CROI consists of the following rules and operations:

Rules:
- `IFC2CROI`: transforms an `IFCModel` to a CROI (single rule; the rest is handled by operations)

Operations:
- `handleIfcObjects`: iterates over all `IfcObjects`, adds their properties and relationships, and appends them to the CROI
- `handleIfcRelationships`: iterates over all `IfcRelationships` of an `IfcObject` and handles them either as `IfcPropertySet` or as `IfcTypeObject`
- `handleIfcPropertySets`: adds `IfcPropertySetDefinition`-specific information according to the CROM, e.g. `hasProperties` in the case of an `IfcPropertySet`, and appends the role to the corresponding `ObjectDefinition`

- `handleIfcTypeObjects`: adds `IfcTypeObject`-specific information, e.g. `hasPropertySets`, according to the CROM and appends the role to the corresponding `ObjectDefinition`
- `handleViewInformation`: adds additional view information if given
- `mergeIfcPropertySets`: merges `IfcPropertySets` according to the property assignment given by IFC; checks if the property values should be taken from the type or if they will be overwritten by a property set on the occurrence
- `mergeIfcProperties`: helper function to implement the operation `mergeIfcPropertySets`
- `addAttributesForType`: helper function to implement the addition of specific information according to the CROM, e.g. the `quantities` attribute only in the case of an `IfcElementQuantity`
- `addProperties`: helper function to add `IfcProperty` elements
- `addQuantities`: helper function to add `IfcPhysicalQuantity` elements
- `retrieveElementName`: retrieves the name of an element due to the fact that the `name` attribute is often optional in IFC
- `retrieveSpecificationName`: retrieves the specification name for an element because an additional differentiation between `IfcTypeObject`, `IfcPropertySet` and `IfcElementQuantity` by means of prefixes is needed

The result of this final workflow step is a `*.croi` file, which is a simple XML file; an excerpt of a CROI of the simple house can be found in listing 3.7.

Listing 3.7: Excerpt of a CROI of the simple house example (`simple_house.croi`)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<croi>
  <ifcObjects>
    ...
    <IfcDoor globalId="0p88roaDKKG8EzXCthnb8k" name="Tür-001"/>
    ...
  </ifcObjects>
  <objectDefinitions>
    ...
    <objectDefinition name="od8">
      <player referenceId="0p88roaDKKG8EzXCthnb8k"/>
      <plays>
        ...
        <PS_PsetDoorCommon globalId="2zJUaDFHsQRSpKCJOUco7C" name="Pset_DoorCommon">
          <IfcPropertySingleValue name="FireExit" nominalValue="True"/>
          <IfcPropertySingleValue name="IsExternal" nominalValue="True"/>
        </PS_PsetDoorCommon>
        ...
        <T_EingangFlS globalId="1fUq2gX$TzJRjHTccJAOHV" name="Eingang 01 1-Fl 1S 23"/>
      </plays>
    </objectDefinition>
    ...
  </objectDefinitions>
</croi>
```

The created CROI can be considered as a runtime model for an application, and both models together – the CROM and the CROI – can now be used as input for an application, which will be further explained in section 3.2.

Altogether, IFC-R consists of three steps: (1) an intermediate transformation from IFC to IFCModel, (2) an M2M transformation from IFCModel to CROM (IFC2CROM), and (3) an M2M transformation from IFCModel and CROM to CROI (IFC2CROI). Before a prototypical implementation applying IFC-R will be explained, the following section will touch on some additional aspects with respect to the developed solution.

Addendum to IFC-R

To the best of my belief, IFC-R is the first role-oriented approach for improving IFC. This proof of concept shows that it is possible to apply the idea and nature of roles to the Industry Foundation Classes. IFC-R has been developed as an additional layer in the current IFC application stack to enrich applications with the advantages of roles. Furthermore, the developed solution provides a simple (domain) view mechanism that will be explained in the upcoming paragraph. Besides, the back transformation from CROM/CROI to IFC will be outlined, and the current status of the implementation will be commented with respect to known issues.

Domain Views As mentioned in chapter 2, the handling of property sets and the concept of object typing lead to various issues. IFC-R tries to improve these shortcomings by utilizing a workflow which transforms IFC models into CROM/CROI models, which achieve better runtime environments and improve the dynamicity in applications. Additionally, this transformation workflow offers hooks for various extensions that create space for further improvements. This has been tested as a part of this proof of concept as well; the result of this is a simple mechanism which allows to apply additional functionality to the preprocessing step (from IFC to IFCModel). For example, filters can be applied to create a specific view of the model.

The method `_apply_view_extensions` in the script `create_ifc_model.py` (listing B.1; starting at line 90) serves as such a hook to add methods to the first workflow step. For example, the method in listing 3.8 aggregates `NominalValues` of a certain `IfcPropertySingleValue` by name. This method can be applied to create an intermediate IFCModel for cost estimations because it offers the possibility to sum up all properties with the same name, e.g. `Price`.

Listing 3.8: Method for aggregating values (view extensions)

```python
def vi_aggregate_numeric_ifc_property_single_value(self, key, ifc_property_name):
    aggregated_value = 0
    for ifc_property in self._ifc_file.by_type('IfcPropertySingleValue'):
        if ifc_property.Name == ifc_property_name:
            value = ifc_property.NominalValue.wrappedValue
            if value.isdecimal():
                # Handle the value as simple integer.
                value = int(value)
            elif self._is_float(value):
                # Handle the value as float.
                value = float(value)
            aggregated_value += value
    return key, str(aggregated_value)
```

The entire script that contains all developed examples can be found in the appendix (listing B.4). Each method in this script can be applied in the preprocessing step depending on the given configuration. For example, an application for the facility management which is only interested in the furniture, the doors, and the windows could apply the configuration shown in listing 3.9.

Listing 3.9: Example configuration for the facility management

```
{
    "view_name": "FM",
    "remove_ifc_property_sets": {
        "del_ifc_property_sets": ["ArchiCADProperties"]
    },
    "filter_ifc_objects": {
        "ifc_objects_filter": ["IfcFurnishingElement", "IfcFlowTerminal", "IfcWindow"]
    },
    "filter_ifc_property_sets": {
        "ifc_property_sets_filter": ["Pset_DoorCommon"]
    },
    "vi_aggregate_numeric_ifc_property_single_value": {
        "key": "SumOfFurniture",
        "ifc_property_name": "Price"
    }
}
```

The corresponding IFCModel would only contain windows (`IfcWindow`), doors (due to a filter for `IfcPropertySets` with the name `Pset_DoorCommon`), furniture (`IfcFurnishingElement`), bathing facilities (`IfcFlowTerminal`), and the sum of the furniture prices, e.g. `<viewInformation value="6326.86" key="SumOfFurniture"/>`.

To put it in a nutshell, the first transformation step in the IFC-R workflow offers a hook mechanism to provide some kind of framework extensibility. This mechanism can be used for further adaptations regarding a specific context such as showing only stakeholder-specific information. Additionally, this information will also be transferred to the CROM/CROI models.

CROM/CROI to IFC Interoperability is an essential characteristic in the area of BIM. Therefore, it is necessary that IFC-R models (CROM/CROI) can be transformed back into IFC models. However, the implementation of such a back transformation would have exceeded the time frame of this thesis; thus, the required tasks[4] will be outlined here.

First of all, most of the transformation rules need to be inverted, e.g. the Natural Type `IfcDoor` needs to be transformed into its corresponding IFC element (also named `IfcDoor`). Due to the fact that `IfcTypeObject`, `IfcPropertySet`, and `IfcElementQuantity` elements have all been transformed into Role Types, it is necessary to implement more logic for this reversion. However, the elements have been provided with a prefix (`T_`, `PS_`, and `EQ_` respectively) to facilitate this task.

Furthermore, the compartments `ObjectDefinition` and their matching `player-/plays`-relations need to be resolved into individual, objectified relationships of the correct type, e.g. `IfcTypeObject` elements need to be attached via the `IfcRelDefinesByType` relationship to an `IfcObject`.

At last, elements which have been added after the transformation workflow should be handled with care. These additional elements should be provided with a valid `GlobalId` and must be mapped to elements of IFC or to an appropriate representation if they do not already exist in any IFC schema. This might be the most complex task in the back transformation, but it is necessary because IFC-R offers the possibility to add naturals and roles away from IFC.

Considering these tasks, implementing an adequate back transformation might seem difficult, however, it is necessary to fulfill the requirement of interoperability with respect to the

[4]This outline of tasks for a proper back transformation has been created with no claim for completeness.

idea of BIM.

Current State and Known Issues IFC cannot be changed easily from scratch, hence, it is necessary to discuss the impact of this circumstance on IFC-R. In order to fully contribute to the idea of BIM, the development has been strongly tied to IFC. To be precise, a strong focus on the existing concepts and a potential back transformation in mind hampered the development in general. However, without this circumstance the capabilities of a role-oriented approach such as IFC-R could have unfold more.

Secondly, many simplifications had to be made during the development of this proof of concept. For example, many data types that have been developed for IFC to improve the semantic meaning (e.g. `IfcLabel`, which is a string for naming purposes) were omitted. Moreover, formal propositions defined by the `WHERE` clauses have been omitted as well because, currently, there is no possibility the represent such rules in CROM/CROI.

Lastly, many of the used libraries and especially CROM have not matured yet. Some wishful features are missing and the development might take a while. Notwithstanding, the current state is sufficient enough for this first proof of concept, which will be demonstrated with the help of prototypes applying IFC and IFC-R in the upcoming section.

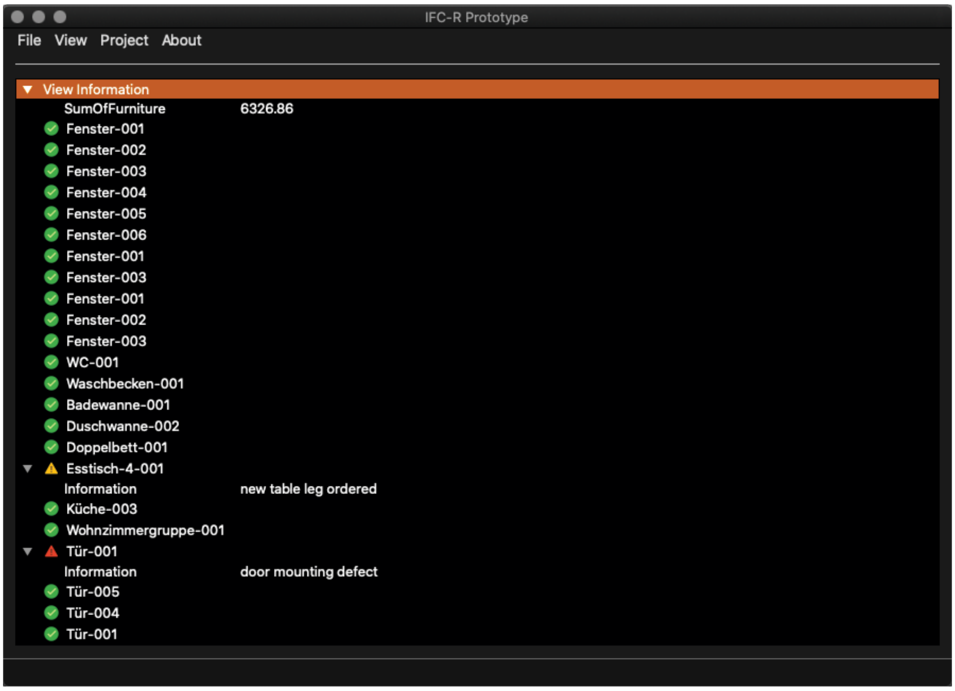
3.2. Prototypical Implementation applying IFC-R

IFC enables modelers to create rich models in the building industry that can be used as the underlying data model for applications in various contexts; IFC-R – as an improved version of IFC – is able to do the same. However, with all its simplifications and in its current state, only basic modeling elements like `IfcObject` and its subclasses are available. Nevertheless, this is sufficient enough to show that IFC-R can be utilized as a data model for applications analogously to IFC, which is why two prototypes have been developed in order to demonstrate that the former can be a proper replacement for the latter.

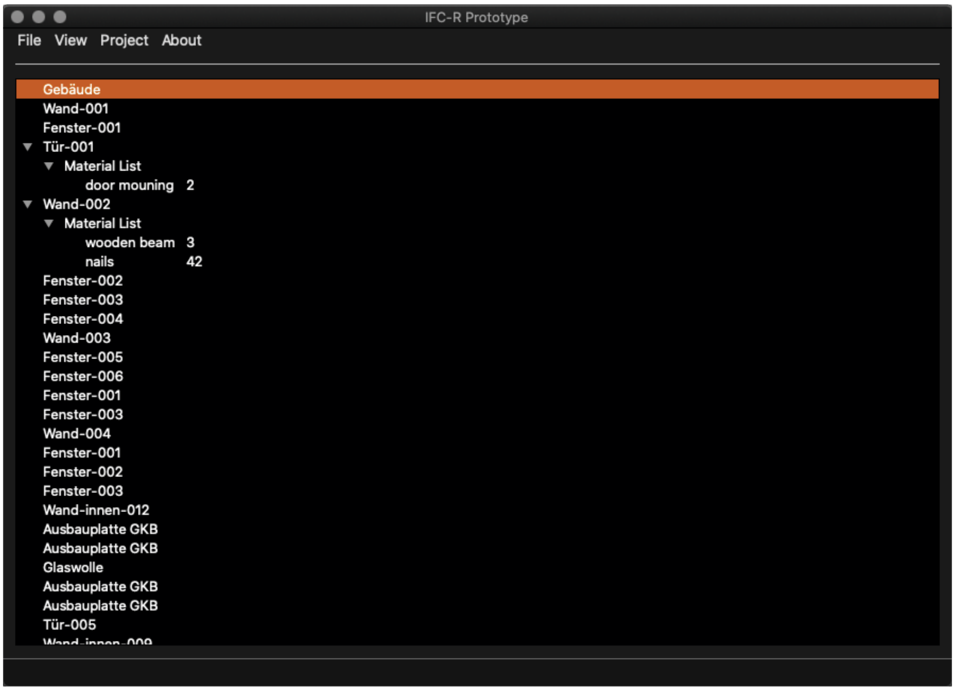
Two prototypes with a graphical user interface (GUI) have been developed. One uses IFC as the underlying data model and the other one uses IFC-R. Again, both prototypes have been developed to mimic an application for the **facility management** and **expense budgeting** (project planning). The prototypes share the GUI, and both use cases will be reflected by the same application. Figure 3.8 shows the general GUI of both prototypes together with examples for the facility management (figure 3.8a) and for the project planning (figure 3.8b). A detailed user manual will be given later.

The GUI was developed with the help of *Qt Creator*[5], and the prototypes have been implemented with Python and the library *PyQt5*[6]. The source code for the data model handling for both prototypes can be found in the appendix (IFC: listing B.5; IFC-R: listing B.6). The script for the general handling of the GUI application has been omitted because it is not necessary for either the explanation of the prototypes nor for the proof of concept of IFC-R.

[5] *Qt Creator* has been used in the version 4.12.4 [The20].
[6] *PyQt5* has been used in the version 5.15.0 [Tho20].



(a) IFC-R prototype showing information for the facility management



(b) IFC-R prototype showing information for the project planning

Figure 3.8.: Different views of the GUI prototypes (own screenshots)

In general, the prototypes offer the following features:

- Loading the particular data model (either an `*.ifc` file, or a `*.crom` and `*.croi` file)
- Saving changes of the models (will be stored in the loaded files)

- Initializing the tree view in the GUI prototypes by adding the particular `IfcObject` elements
- *Adding and removing `IfcPropertySet` elements*
- *Adding and removing `IfcProperty` elements (to/from an existing `IfcPropertySet`)*
- *Changing the value of `IfcProperty` elements*
- *Searching for an `IfcPropertySet` (inside a given `IfcObject`) or an `IfcProperty` (inside a given `IfcPropertySet`) element by name*
- Providing the information for the facility management or the project planning view of the applications

The features written in italics will be offered indirectly by means of dedicated methods like marking an element in the tree view as broken for the facility management. The applications do not offer the direct manipulation of `IfcPropertySet` or `IfcProperty` elements.

Additionally, the IFC-R prototype offers the following features indirectly – again, the functionality cannot be called directly via the application – due to special handling of the CROM (metamodel):

- Initializing the CROM
- Checking if a metamodeling element does already exist in the CROM
- Checking if a given role is played by any of the existing players
- Finding the corresponding `ObjectDefinition` for a given player
- Finding the corresponding `ObjectDefinition` for a given role
- Adding and removing roles from the CROM
- Adding `ObjectDefinition` elements to the CROM
- Showing the (domain) view information if given (additional feature of IFC-R)

The prototypical implementation itself is not very complex. The data models are processed in-memory in the applications, handling of IFC elements has been simplified – e.g. mainly working with String representations of properties – and general features such as input validation have been reduced to a minimum.

Before the comparison of the prototypes or their underlying data models can be made, the usage of the applications will be described in the next section.

User Manual

In order to understand the previously mentioned features of the applications, this section will give an overview on how to use the prototypes.

Menu The menu offers various options for working with the applications:

File: General loading and storing options
- **Load**: Opens the file dialog (OS-specific) for loading the input for the corresponding application
  - **IFC**: Input is an `*.ifc` file (STEP file)
  - **IFC-R**: Input is a folder/directory containing `*.crom` and `*.croi` files
- **Save**: Persists changes of the models in the loaded files

View: Toggles information in the tree view of the application (for both use cases)
- **Show maintenance status**: Shows an icon indicating the status and a stored comment (if available) of each object

– **Show material lists**: Shows a list of material information on each object (if available)

Project: Information for the project planing use case (expense budgeting)

– **Order material**: Opens the dialog for simulating an order process

About: Opens a dialog containing copyright information

Selecting a folder/directory in the IFC-R prototype opens a follow-up dialog (figure 3.9), which lists all found CROM- and CROI-files. In its current state, the prototype does not check if the selected models match – selecting mismatching files could possibly crash the application.

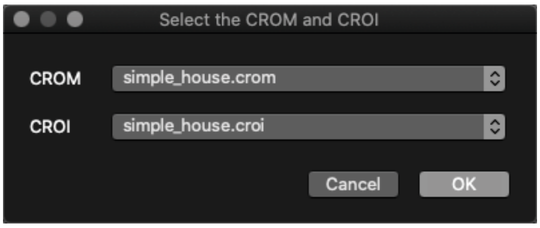


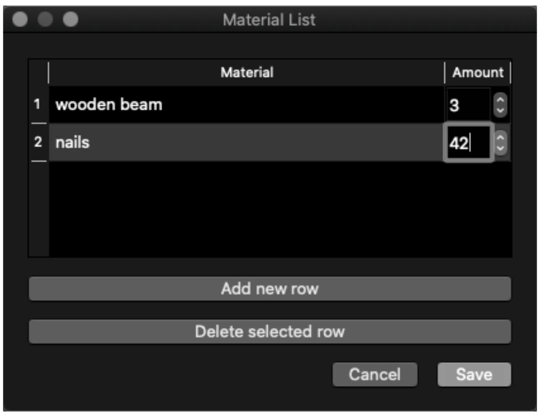
Figure 3.9.: Dialog for selecting a particular CROM and CROI (own screenshot)

Main Window The main window contains the tree view that lists all `IfcObject` elements by their `Name` attributes, e.g. an `IfcDoor` named `Tür-001` after the input files have been successfully loaded and processed. Right-clicking an item in the tree view opens a context menu:

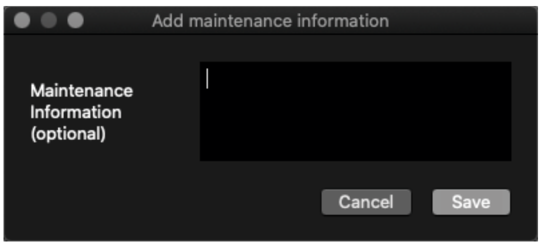
Edit material list (project planning/facility management): Opens the dialog for adding material to the object's material list (figure 3.10a)

Mark as broken (facility management): Marks an object as broken; this opens a dialog for adding additional information (figure 3.10b) and changes the context menus entries as follows:

– **Mark as fixed**: Marks the object as fixed, e.g. after a direct repair (an additional comment can be submitted)
– **Mark as pending**: Marks the object as pending; simulating that the damage of the object has been noticed but, for example, spare parts need to be ordered first



(a) Dialog for adding material to the material list of an object



(b) Dialog for adding an optional information to the corresponding maintenance status (`broken`, `pending`, or `fixed`)

Figure 3.10.: Dialogs of the main window (own screenshots)

The information about the maintenance status (icons and comments) and the material lists are only visible if the corresponding option has been selected via the “View”-menu.

Ordering Process If at least one object has a maintained material list, the ordering process can be started. This opens a new dialog (figure 3.11) which enables the simulation of such an ordering process. The listed material can be ordered completely or just a certain amount of it.

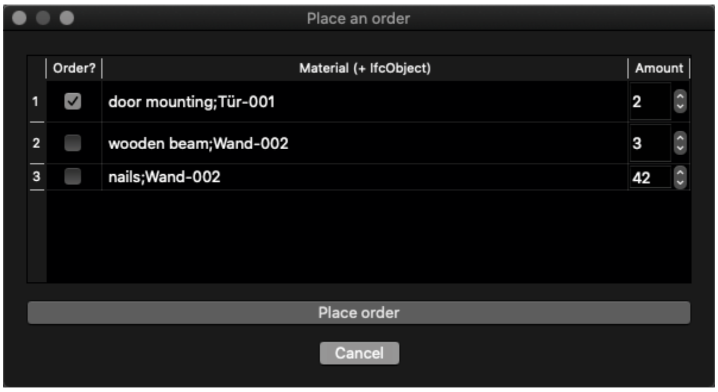


Figure 3.11.: Dialog for the simulation of an ordering process (own screenshot)

Each aforementioned feature stores the necessary information in an `IfcPropertySet` or an `IfcPropertySet`-role respectively. For example, the maintenance information is stored in an `IfcPropertySet` named `Maintenance` with the following `IfcPropertySingleValue` elements: `Status` and `Information`. As mentioned above, the scripts work with an in-memory representation of the loaded input files in order to keep the prototypes as simple as possible. This representation will be persisted upon a click on “File” > “Save”.

Despite the simplicity of the prototypes, both applications offer the same functionality for their end-users. However, highlighting the full advantage of a role-oriented approach such as IFC-R with a simple implementation and such small-scale use cases is difficult. Moreover, it is possible that a direct comparison of the prototypes themselves does not show whether IFC-R performs better than IFC. Therefore, an in-depth comparison of the underlying data models is crucial for answering RQ2.

4. Comparing IFC and IFC-R

Following the elaboration of the proof of concept and the implemented prototypes, this chapter concentrates on comparing the developed solutions. To be precise, the comparison will not cover the two prototypes themselves but rather the underlying data models, namely IFC and IFC-R. In order to achieve a better comprehension and replicability of the evaluation, section 4.1 will introduce the used software metrics and the metrology. Section 4.2 will then cover how to apply the measurements and will present the results for both data models. Certain problems and limitations concerning the metrology will be discussed in 4.3 in order to understand and classify the results of the evaluation. Along with that, an interim conclusion and the answer to the second research question will finish this chapter.

4.1. Definition of used Software Metrics

There are many ways to comprehend software. In order to deeply understand an application or software system, one can analyze code either statically or dynamically, study available documentation, or talk to the developers, who have expert knowledge of the subject [Sne95]. A fifth approach, which will be the foundation of this comparison, is software comprehension by means of metrics, that means understanding aspects of the software through numbers [Sne95].

The first attempt to compare both approaches was the application of typical software metrics like Cyclomatic Complexity (CC) [McC76; Abr10] and *Halstead*'s metrics [Hal77; Abr10] to acquire comparable values. The Python tool `radon`[1] was used to determine the values for these two metrics and the so called Maintainability Index, which derives from the CC, the *Halstead* Volume and the Source Lines of Code (SLOC) [Lac20]. Table 4.1 shows an excerpt of the measured values of both prototypes according to these metrics. This will be subject of the next paragraphs.

[1] *Radon* (used in Version 4.1.0) is a Python tool to apply typical software metrics to Python scripts. See [Lac20] for further information.

Table 4.1.: Comparison of prototypes with typical software metrics (with `radon` for Python)

Software Metric	Prototype with IFC	Prototype with IFC-R
Cyclomatic Complexity (average complexity)	A (3.0)	A (3.48)
Halstead's Metrics (selected metrics)	Volume: 366.85 Difficulty: 3.5 Effort: 1283.98 Bugs: 0.122	Volume: 1020.88 Difficulty: 5.49 Effort: 5603.75 Bugs: 0.34
Maintainability Index (derived metric)	A	A
Source Lines of Code (raw metric)	134	250

The Cyclomatic Complexity originated as a metric in the graph theory and, understandably, has been used by McCabe to measure the "complexity" of a control flow graph (CFG). This abstract representation of a single module in a program is the basis for the so called "McCabe number", which will be acquired by simply counting the number of edges and nodes in such a CFG [Abr10, p. 135]. McCabe defined the following equation:

$$v(G) = e - n + 2 \tag{4.1}$$

in which $v(G)$ defines the cyclomatic number, e the number of edges and n the number of nodes in a CFG G for a single module [Abr10, p. 135]. The calculated number $v(G)$ can be used for estimations, e.g. determining the number of paths for testing in order to reach a certain test coverage, or interpretations such as the bigger the number the harder it will be for a developer to understand the overall logic of a module.

As described in chapter 3, both prototypes have a low complexity in general. Each prototype operates on the in-memory data structure of its input file(s). There is also no complex creation of objects nor heavy methods with lots of loops and statements, hence, the CC is generally low and the values in table 4.1 can be rated as no significant difference.

However, the *Halstead*'s metrics indeed do look like there is a meaningful difference. This can be explained by elaborating on what these metrics measure. Halstead characterizes an implementation of a method or an algorithm as a set of tokens, which can be categorized as **operators**, for example a function, and **operands**, like variables and specific values [Abr10, p. 146]:

- η_1: Number of distinct operators
- η_2: Number of distinct operands
- N_1: Total number of occurrences of operators
- N_2: Total number of occurrences of operands

After counting these tokens the program length N and the program vocabulary η can be calculated using the following equations [Abr10, p. 147]:

$$N = N_1 + N_2 \tag{4.2}$$

$$\eta = \eta_1 + \eta_2 \tag{4.3}$$

which can in turn be used to derive metrics like Volume and Difficulty [Abr10, pp. 147 sq.]. Consequently, that means the measured differences are based on a difference in the number of used operators and operands, which in fact is a result of an overhead due to additional handling to keep the CROM consistent with the CROI. This difference also applies to the values for the Cyclomatic Complexity and the SLOC and can also be considered as non-significant.

Analyzing the metrics and values revealed that there is (a) no significant difference between both prototypes, and (b) these typical metrics are not suitable for the comparison of IFC and IFC-R. This results from a focus on the structural features of a program, like the control flow or the number of used variables, instead of the underlying data models, which is needed in order to evaluate the intended goal of IFC-R. That is why other measurement methods need to be identified. However, the measured values provide an indication that IFC-R can be used in applications without a significant loss in quality and only a slight overhead.

4.1.1. Identifying suitable measurement methods

The term “software measurement” has matured over the past years [Pfl08; Abr10] and is now an important aspect for both practitioners and researchers. However, in contrast to other sciences, there is insufficient consensus on specific measurement methods and software metrics. As a consequence, a plethora of different methods and metrics in the field of software engineering emerged [Abr10]. That is why identifying suitable ways to properly measure or rate IFC and IFC-R is difficult.

The approach for this thesis follows the three steps defined by Abran in [Abr10, pp. 21 sqq.], i.e. (1) selecting a measurement method or designing a new one, (2) applying its rules to measure the data models, and (3) evaluating the acquired results in order to compare both models [Abr10, p. 21]. The first step is subject of this section, step two and three will be considered in section 4.2 subsequently.

According to Abran, the first sub-step towards a suitable measurement method is the “Determination of the measurement objectives” [Abr10, p. 25]. With respect to the intended goal of this thesis, the **measurement objectives** are extensibility and maintainability, more precisely, if IFC-R is able to improve these aspects compared to IFC from the perspective of modelers and developers.

The second sub-step focuses on the “Characterization of the concept to be measured” [Abr10, p. 25], i.e. identifying the measurand. This refers to the **entities** to be measured with respect to characteristic **attributes** [Abr10, p. 25]. To prevent further misuse of the concepts of IFC and the creation of new workarounds, an improvement should have the following characteristics:

- an easier way to extend IFC
- improved maintainability for existing and new classes
- comprehensible concepts for modelers and developers
- no loss in quality or performance

The third sub-step is important for determining the relationship between the abovementioned criteria and the specific attributes to be measured [Abr10, pp. 27 sq.]. This is necessary to ensure that the measured entities do reflect the intended goal otherwise a proper evaluation would not be possible. Hence, the evaluation will be strongly tied to specific use cases with respect to the attributes of the underlying data models, e.g. taking into account how many classes or objects will be needed to express the same information. As noted by Abran, the second and third sub-step are strongly connected to each other, which means that these will be typically performed together [Abr10, p. 28].

The fourth and last sub-step consists of the “Definition of the numerical assignment rules” [Abr10, p. 25], i.e. specifying units and scales and how these numbers and values will be acquired. This will be explained in detail after defining the selected measurement method.

To put it in a nutshell, the entities to be measured will be the data models IFC and IFC-R with respect to certain use cases. These use cases will be viewed from two perspectives (1) the modelers working with IFC or IFC-R, and (2) the developers for applications employing the data models. Furthermore, it is important to consider the technical aspects of both approaches and the people who will work with them. Therefore, the so called Use Case Points (UCP) method by Karner [Kar93] will be applied. This method was developed to measure the functional size of software based on the requirements that can be defined through use cases. The intended goal is an estimation for a software project derived from the analyzed size and complexity with respect to technical aspects, e.g. if the system is distributed, and environmental aspects like the developers’ experience [Kar93; Abr10; ONK11]. Although the intent of this method was not to be used for the comparison of two data models, its nature of considering technical and environmental aspects makes it a suitable metric for comparing IFC and IFC-R.

4.1.2. The Use Case Points (UCP) method

Karner developed the Use Case Points method in 1993 in order to measure the functional size of software in its early development stage [Abr10]. The original measurement method consists of the following parts [Kar93; Abr10; ONK11]:

Unadjusted Use Case Weight (UUCW) The UUCW is an assessment of the complexity of the given use cases. Initially, each use case will be categorized by counting the number of transactions and will receive a corresponding weight [ONK11]:

- *simple*: less than four transactions → **weight: 5**
- *average*: between four and seven transactions → **weight: 10**
- *complex*: more than seven transactions → **weight: 15**

After that categorization, the use cases of each category will be summed up and multiplied by their weights resulting in the following equation [ONK11]:

$$UUCW = \#SimpleUseCases \times 5 + \#AverageUseCases \times 10 + \#ComplexUseCases \times 15 \quad (4.4)$$

Unadjusted Actor Weight (UAW) The participating actors, besides the use cases themselves, are another factor for calculating the UCP. Therefore, a similar categorization of the actors’ complexity is the next step and results in the UAW. The categorization of actors depends on their characterization according to the following predefined types [ONK11]:

- *simple*: a system working with an API → **weight: 1**
- *average*: a system working with a protocol like HTTP *or* user interacting with a terminal console → **weight: 2**
- *complex*: an actor working with a system through a GUI → **weight: 3**

Calculating the UAW is also similar to the UUCW resulting in the following equation [ONK11]:

$$UAW = \#SimpleActors \times 1 + \#AverageActors \times 2 + \#ComplexActors \times 3 \tag{4.5}$$

Afterwards, the **Unadjusted Use Case Points (UUCP)** can be calculated using the following equation [ONK11]:

$$UUCP = UAW + UUCW \tag{4.6}$$

This will be adjusted by taking the **technical** and **environmental** aspects into account.

Technical Complexity Factors (TCF) Karner defined 13 technical factors[2] with a specific weight. Each of them should be evaluated and provided with a value from zero (meaning irrelevant) to five (meaning important). The total influence of these factors can then be calculated using the following equation [ONK11]:

$$TCF = 0.6 + (0.01 \times \sum_{i=1}^{13} WeightOfTechnicalFactor_i \times GivenValue_i) \tag{4.7}$$

Environmental Factors (EF) According to Karner, the given environment, such as the motivation of a team, should be considered for the estimation as well. As a result, he defines additional eight environmental factors[2], which will be evaluated and calculated similar to the TCF resulting in the following equation [ONK11]:

$$EF = 1.4 + (-0.03 \times \sum_{i=1}^{8} WeightOfEnvironmentalFactor_i \times GivenValue_i) \tag{4.8}$$

Finally, the UCP can be calculated by multiplying all acquired values [ONK11]:

$$UCP = UUCP \times TCF \times EF \tag{4.9}$$

In order to acquire a final effort estimation, the calculated UCP must be multiplied by a so called productivity factor. However, for the comparison of IFC and IFC-R this final estimation number is not necessary because the Use Case Points method is only used to make the data models comparable by means of numbers. Additionally, certain criticism by Abran regarding the metrology of the UCP method [Abr10, pp. 192 sqq.] and some adjustments according to Ochodek, Nawrocki, and Kwarciak from [ONK11] will be discussed next.

4.1.3. Adapting the UCP method

As mentioned beforehand, the purpose of using the UCP method is to make both models comparable. However, in order to achieve this goal, the method needs some adaptations.

[2] They will not be covered in detail because of certain adaptations according to [ONK11] (See section 4.1.3).

First of all, the complexity of the use cases will be equal because IFC and IFC-R will be compared considering the same use cases for the exact same actors. Therefore, the value for UUCW will be considered as a constant. Furthermore, a study by Ochodek, Nawrocki, and Kwarciak revealed that the impact of UAW is minor [ONK11] and, hence, will be omitted. This is why the equation (4.6) for the UUCP will be simplified as follows:

$$UUCP = \underbrace{UAW}_{0} + \underbrace{UUCW}_{1} = 1 \tag{4.10}$$

Secondly, Ochodek, Nawrocki, and Kwarciak examined that the 13 technical factors defined by Karner do overlap, which means that they can be grouped together to create more general factors. The same applies to the eight environmental factors, which results in the following reduced set for the TCF [ONK11]:

- Efficiency
- Operability
- Maintainability
- Interoperability

and EF [ONK11]:

- Team Experience
- Team Cohesion

Since the usage of IFC respectively IFC-R will be considered from the viewpoint of single actors only, the environmental factor *Team Experience* will be simplified to *Experience* (of the specific actor) and *Team Cohesion* will be omitted. Moreover, the entities to be measured are the data models, which means that no software system needs to be operated directly. Therefore, *Operability* will be substituted with an important characteristic of working with models, namely *Understandability*. In addition to that, both data models are able to create the same output format, namely an XML file. *Interoperability*, consequently, is no important aspect and will be exchanged by a main property of this thesis: *Extensibility*.

Thirdly, the technical and environmental factors have been evaluated according to the influence each factor will have on a project, whereas the comparison should reflect upon to which degree each factor will be accomplished by the corresponding data model. The scale, hence, will be modified in order to fulfill this need. That means, a value of zero expresses no accomplishment. In contrast, a value of five means that this aspect needs no further enhancement.

Lastly, Abran and Ochodek, Nawrocki, and Kwarciak have criticized the high degree of subjectivity while calculating the individual parts of the UCP method [Abr10; ONK11]. Especially in [ONK11], the difference between two people applying this method was not negligible. In consequence, the degree of subjectivity will be reduced by supporting the decisions with additional metrics defined in [Sne95], [RH97] and [ISO15], and a set of own metrics.

4.1.4. Supporting Metrics

With respect to the previously defined technical and environmental factors, the evaluation of these should be supported by further metrics. This is why each factor will be assigned at least one additional metric if possible, which will be further elaborated on in this paragraph.

Efficiency The term "Efficiency" in this context refers to a proper performance in relation to the given resources [ONK11]. To be precise, a modeler should be able to employ IFC or IFC-R with ease in order to develop a performant model independent from the complexity of the given task [ONK11]. Furthermore, a developer implementing an application utilizing the data models should be able to easily understand the concepts and to estimate needed resources.

With respect to the evaluation of the **Efficiency**, the first supporting metric will be Coupling Between Object Classes (CBO) by Rosenberg and Hyatt defined in [RH97]. This metric "is a [simple] count of the number of other classes to which a class is coupled" [RH97]. Each distinct related class (hierarchy) will be counted, meaning that inheritance related coupling will be ignored. This metric can be defined as the following equation:

$$CBO = \#CoupledClasses \tag{4.11}$$

A high degree of coupling means that changes to this class can influence any other coupled class, which then results in a bad modularity hampering reuse and also decreasing maintainability [RH97]. Understandability will also be reduced due to more classes being involved in a task, which increases the number of relations to keep in mind. As a result, efficiency can get decreased.

Additionally, the runtime of an application will be considered. Therefore, the possible number of runtime objects will be counted. This metric will be called Runtime Efficiency (RE). It can be partly derived from the CBO and results in the following equation:

$$RE = \#PossibleRuntimeObjects \tag{4.12}$$

The more objects are created during runtime, the more memory is needed in order to process the data model efficiently.

Understandability The design of a (meta)model should be easily understood by each actor working with it. This characteristic is known as **Understandability** and can be considered as a part of efficiency. However, in order to compare IFC and IFC-R it is important to analyze the data models' expressivity and if the corresponding concepts are semantically correct. Unfortunately, this is hard to measure but will be made perceivable by employing the abovementioned CBO and the Data Model Understandability (DMU) defined in [ISO15]. This puts the number of elements considered "understandable" in relation to the number of elements provided by a data model and can be defined as follows:

$$DMU = \frac{\#UnderstandableElements}{\#Elements} \tag{4.13}$$

If each provided element is understandable, DMU is equal to one, which is perfect.

In addition to that, the so called Control Flow Complexity (CFC) defined by Sneed in [Sne95] will be used. The more nodes a control flow needs to pass, the more complex it is, lowering the level of understandability for a developer. This metric will be applied with regard to (test) code snippets utilizing the data models.

$$CFC = \frac{(\#Edges - \#Nodes + (2 \times \#Procedures))}{\#Statements} \tag{4.14}$$

In contrast to Sneed, this metric will only be applied to those code snippets and will not cover any of the prototypes because their main purpose is proofing that IFC-R can be a replacement for IFC for the use cases defined in chapter 3.

Maintainability As described earlier, one of the main goals of this thesis is the improvement of the **Maintainability** of IFC. That means, it is important that a (meta)model can be modified easily and certain concepts or code can be reused [ONK11]. In order to achieve this, the data models have to be analyzed regarding their structure since the more complex a model, the harder it will be to maintain. Therefore, the CBO metric can be reused in order to evaluate this aspect as well.

Maintaining a model also means being able to append and remove properties. This feature will be further evaluated from the viewpoint of a developer, more precisely taking the complexity of such a workflow into account. Therefore, the ratio of conditional statements in relation to the total number of actions in a UML activity diagram will be measured, resulting in the equation for the Workflow Complexity (WC):

$$WC = \frac{\#DecisionBlocks}{\#ActionBlocks} \tag{4.15}$$

The closer this value is to one, the more complex such an action will be, resulting in a more complicated maintainability process.

Extensibility Extending IFC has been briefly discussed in chapter 1 and has been examined in depth in chapter 2. Consequently, it is important to evaluate the **Extensibility** of IFC-R compared to IFC. In this context that means being able to extend the standard by means of "new" classes, e.g. the concept of a `FireDoor`. However, quantifying this aspect is hard because a software, system, or model is either extensible or not. Extensibility, nevertheless, depends on characteristics like coupling, which is why CBO can be utilized again. To be precise: the more a class is independent from other classes, the easier it can be extended.

Experience The only environmental factor which will be considered in this evaluation is **Experience**. Unfortunately, this is not measurable in the context of this thesis. Aspects such as the familiarity and understandability of certain concepts are important and could be checked by interviewing modelers and developers. That is why this aspect will not be covered by additional metrics, however, it should be audited in future work.

To sum it up, an adapted version of the Use Case Points method will be applied in order to comprehend both data models and to make them comparable by means of quantification. Nevertheless, the high degree of subjectivity is not negligible and will be discussed in section 4.3. To compensate this issue, a set of supporting metrics has been introduced finalizing the definition of the used software metrics. Table 4.2 gathers all information about the metrology, which will be applied in the upcoming section.

Table 4.2.: Summary of applied software metrics (adapted template by [Abr10, p. 30])

Measurement Objectives	
What will be measured?	The data models IFC and IFC-R
Entities to be measured	Structure and concepts of both models
Measurement **point of view**	Modelers and Developers
Measurable Construct	
Attributes to be measured	Efficiency Understandability Maintainability Extensibility Experience
Measurement Method	
Main **method**	Use Case Points (UCP) method
Additional **metrics**	Coupling Between Object Classes (CBO) Runtime Efficiency (RE) Data Model Understandability (DMU) Control Flow Complexity (CFC) Workflow Complexity (WC)

4.2. Evaluation of IFC and IFC-R

The foundation for the evaluation has been set in section 4.1. After the identification of a suitable measurement method and the definition of additional metrics, this section will now focus on applying the metrology in order to achieve comparable values. That means (1) setting the requirements for the UCP method, (2) applying the supporting metrics and gathering the corresponding values, (3) harnessing these values to perform the UCP method, and (4) evaluating the results with respect to IFC and IFC-R.

Starting with the requirements of the UCP method, the first task will be a conversion of the measurement objectives into use cases. Although the evaluation of the UUCW and the UAW, i.e. measuring the complexity of the use cases and the actors, has been simplified, it is necessary to elucidate these to comprehend the intended goals. For that reason, figure 4.1 illustrates the measurement objectives mentioned in section 4.1.1 in a use case diagram.

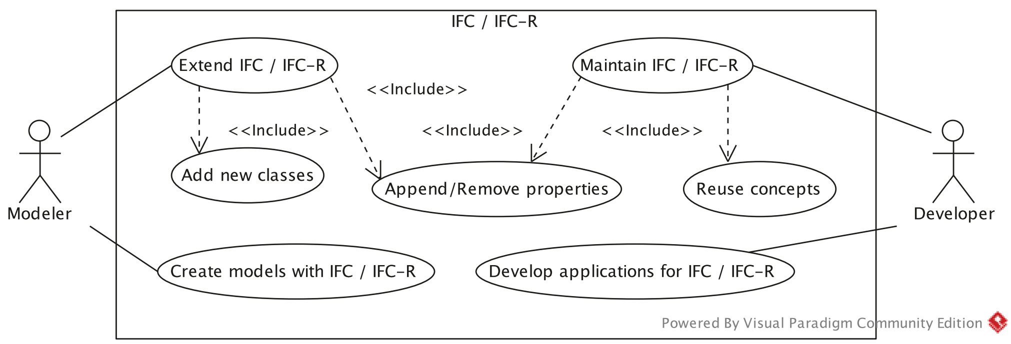


Figure 4.1.: Use case diagram illustrating the examined measurement objectives for IFC and IFC-R

The assessment of these use cases involves the structure and concepts of both data models. That means, from a modelers perspective, how easy each model can be extended and, especially for IFC-R, if models can be created at least as good as with IFC. Furthermore, the viewpoint of developers is associated with the maintainability, which is exemplified by means of appending and removing properties, and an evaluation of the development process for applications utilizing the data models. The illustrated use cases cover the parts of IFC respectively IFC-R analyzed in this thesis, however, in order to fully contribute to BIM, further important use cases need to be assessed in future work.

4.2.1. Gathering the supporting metrics

In order to sustain the assessment of the data models during the application of the UCP method, section 4.1.4 introduced supporting metrics. Hence, these metrics will be applied to gradually collect their values in this section.

Coupling Between Object Classes The first value will be measured through Coupling Between Object Classes (CBO). As described in the corresponding section, subject of this metric is the degree of coupling in the class hierarchy. However, due to the size and dimensions of the IFC standard, and in order to achieve a proper comparison only an excerpt of its hierarchy will be considered. To be precise, only the parts which have a corresponding equivalent in IFC-R, as described in chapter 3, will be taken into account. As a result, the class diagrams[3] in figure 4.2a and 4.2b were created as a basis for counting.

Counting includes all classes which are referenced in the examined class and, if existing, methods called from other classes. However, as described in chapter 2 there are no methods in IFC classes and, therefore, none are available in IFC-R. This results in the following values for CBO in figure 4.2a:

$$CBO(\texttt{IfcObject}_{\text{IFC}}) = 2$$
$$CBO(\texttt{IfcTypeObject}_{\text{IFC}}) = 2$$
$$CBO(\texttt{IfcPropertySet}_{\text{IFC}}) = 3$$

Adding up all values will result in a final CBO for this use case:

$$\underline{CBO(\text{IFC})} = CBO(\texttt{IfcObject}_{\text{IFC}}) + CBO(\texttt{IfcTypeObject}_{\text{IFC}}) + CBO(\texttt{IfcPropertySet}_{\text{IFC}}) = \underline{7}$$

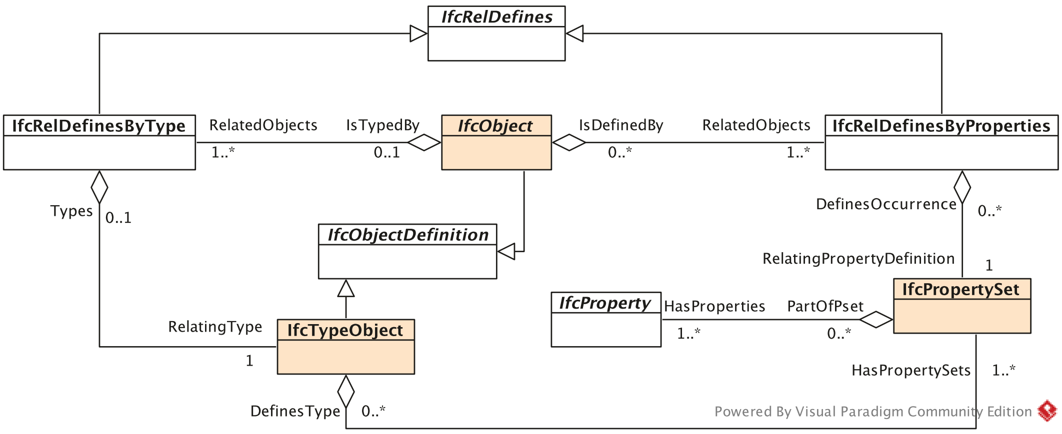
Counting the classes in 4.2b is slightly different because the CBO metric is usually applied to object-oriented models. The foundation of IFC-R, however, is role-oriented modeling. Therefore, the premise for this use case is that, beside normal classes, each `CompartmentType`, `RoleType` and `DataType` will be counted as a class if they are referenced. This simplification leads to the following values:

$$CBO(\texttt{IfcObject}_{\text{IFC-R}}) = 3$$
$$CBO(\texttt{IfcTypeObject}_{\text{IFC-R}}) = 2$$
$$CBO(\texttt{IfcPropertySet}_{\text{IFC-R}}) = 3$$

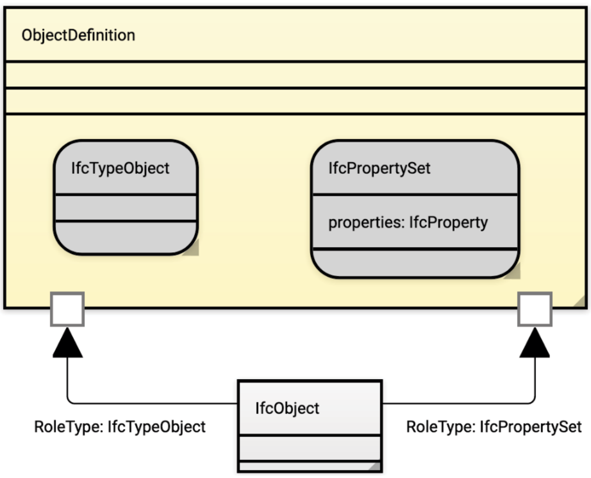
[3] The "class" diagram for IFC-R uses the graphical notation of CROM and, thus, is no actual class diagram as defined in the UML. However, it will be synonymously referred to as class diagram in order to keep up the reading flow.

Resulting in the final value:

$\underline{CBO(\text{IFC-R})} = CBO(\texttt{IfcObject}_{\text{IFC-R}}) + CBO(\texttt{IfcTypeObject}_{\text{IFC-R}}) + CBO(\texttt{IfcPropertySet}_{\text{IFC-R}}) = \underline{8}$



(a) Class hierarchy in IFC



(b) Class hierarchy in IFC-R

Figure 4.2.: Class diagrams for IFC and IFC-R with focus on the classes `IfcObject`, `IfcTypeObject` and `IfcPropertySet`

Runtime Efficiency As a second step, the number of possible runtime objects will be counted for the Runtime Efficiency (RE). As mentioned above, this number can be partly derived from the CBO. Thus, figure 4.2 will be the basis as well. However, counting the number of runtime objects depends on specific runtime environments. More precisely, the actual number differs with conditions such as the number of occurrences in a model. Hence, the following set of attributes serves as an example in order to derive values for RE:

- a class `IfcDoor` (inheritance hierarchy of `IfcObject`)
- defined by one `IfcTypeObject` containing one `IfcPropertySet` with three `IfcProperties`
- further defined by two `IfcPropertySets` with one `IfcProperty` each

Starting with IFC, a naïve implementation of this example would produce one object for each occurrence mentioned. Moreover, each relationship (`IfcRelDefines`) between `IfcObject` and an `IfcTypeObject` or `IfcPropertySet` produces an additional object. This example results in the following value:

$$\underline{RE(\text{IFC})} = \underbrace{10}_{\text{from the example}} + \underbrace{3}_{\text{for each relationship}} = \underline{13}$$

An implementation utilizing IFC-R – under the same premise – would also produce one object for each occurrence mentioned. However, there would be only one additional object produced for the `ObjectDefinition`, resulting in:

$$\underline{RE(\text{IFC})} = \underbrace{10}_{\text{from the example}} + \underbrace{1}_{\text{for the compartment}} = \underline{11}$$

Data Model Understandability Rating the "understandability" for the Data Model Understandability (DMU) is not trivial. What is easily comprehensible for one person could lead to hours of research for others. Furthermore, it is hard to determine if, for example, a concept is hard to understand because of its wording or of its underlying complexity. The counting of understandable elements, therefore, will focus on general concepts, the naming of properties which might be misleading, and the relations between classes. Again, 4.2 will set the scene for this evaluation and a total of five aspects for each data model will be examined.

On the one hand, the concept of objects (`IfcObject`) being further defined by properties (`IfcPropertySet`) is easy to grasp in IFC (+1)[4]. On the other hand, the naming of the attribute holding the properties (`HasProperties`) is rather misleading because the keyword "has" is usually connected to a boolean check (+0).

Secondly, the typing concept (`IfcTypeObject`) can cause confusion. For example, the class `IfcDoorType` holds common properties for doors, which will be added via a relationship. The question to be asked here is: Why can these common properties not be added directly to the class `IfcDoor` (+0)?

Thirdly, the objectified relationships (`IfcRelDefines`) would be reasonable if they provided more information than being the connection between two classes. However, they do not and the semantic meaning of this relationship could have been achieved by proper naming of the corresponding attributes as well (+0).

Lastly, the usage of inverse attributes is a good way to improve the concept of objects being in relation to one another. With respect to the objectified relationships, this improves the understandability to some extent (+1). Adding up all values results in the following:

$$\underline{DMU(\text{IFC})} = \frac{2}{5} = \underline{0.4}$$

On the other hand, the main concept of IFC-R, namely role-oriented modeling, is easy to grasp for many people. Real analogies, like an actress playing a role in a movie, help understand this concept in more detail (+1). However, the necessity of compartments is not easily comprehensible (+0).

In comparison to IFC, the naming of attributes (`HasProperties` to simply `properties`) has been a conscious decision in order to resolve misconceptions due to naming issues (+1).

[4] A label of (+1) or (+0) marks an "element" for counting with respect to the DMU metric.

In addition to that, the objectified relationships have been removed, which means that the `IfcTypeObjects` and `IfcPropertySets` will be attached directly to an `IfcObject` as a role (+1).

Despite the goal of improving IFC, having an additional step in the overall workflow increases the complexity. That means, the transformation of IFC to IFC-R introduces new pitfalls in seeing the big picture, hampering the understandability (+0). Evaluating these five aspects results in the following value for DMU:

$$\underline{DMU(\text{IFC})} = \frac{3}{5} = \underline{0.6}$$

Control Flow Complexity As described in section 4.1.4, the values for the Control Flow Complexity will be acquired with the help of test code snippets. These snippets implement the following tasks:

(*A*) Reading a data model, retrieving the first `IfcDoor` element, and printing the name of the first found `IfcProperty` (IFC: Listing B.7, IFC-R: Listing B.8)

(*B*) Reading a data model and printing the `GlobalId` of each `IfcObject` being defined by an `IfcPropertySet` named `Pset_FireRatingProperties` (IFC: Listing B.9, IFC-R: Listing B.10)

The Python tool *StatiCFG*[5] was used to create control flow graphs (CFGs) as a basis for the CFC metric. Counting the number of edges, nodes, procedures, and statements on the CFGs in figure A.2 and A.4 (in the appendix) for IFC leads to the following values for the CFC:

$$CFC(A_{\text{IFC}}) = \frac{(7 - 5 + (2 \times 1))}{9} = \frac{4}{9} = 0.44$$
$$CFC(B_{\text{IFC}}) = \frac{(11 - 7 + (2 \times 1))}{9} = \frac{6}{9} = 0.67$$

Resulting in the final value:

$$\underline{CFC(\text{IFC})} = \frac{(CFC(A_{\text{IFC}}) + CFC(B_{\text{IFC}}))}{\#Tasks} = \frac{(0.44 + 0.67)}{2} = \underline{0.56}$$

Applying the same metrology for IFC-R on the CFGs in figure A.3 respectively A.5 (in the appendix) results in:

$$CFC(A_{\text{IFC-R}}) = \frac{(14 - 9 + (2 \times 1))}{12} = \frac{7}{12} = 0.58$$
$$CFC(B_{\text{IFC-R}}) = \frac{(7 - 5 + (2 \times 1))}{6} = \frac{0}{9} = 0$$

Finally resulting in:

$$\underline{CFC(\text{IFC-R})} = \frac{(CFC(A_{\text{IFC-R}}) + CFC(B_{\text{IFC-R}}))}{\#Tasks} = \frac{(0.58 + 0)}{2} = \underline{0.29}$$

[5] *StatiCFG* (used in version 0.9.5) is a Python package that can be used to create control flow graphs of Python 3 scripts. See [Coe20] for further information.

Workflow Complexity The last supporting metric applied in this section is the Workflow Complexity (WC). Measuring the complexity of control flows and workflows helps identify critical parts of an implementation or the underlying logic. The degree of complexity will be derived by counting the decision blocks in an UML activity diagram because such abstractions of workflows, by utilizing a graphical notation, can be a guidance to critical spots.

Like the RE and CFC metrics, the counting depends on specific use cases. For that reason, the following typical scenarios while working with IFC will be considered for this metric:

(*A*) Adding an `IfcProperty` to an `IfcObject`
(*B*) Removing an `IfcProperty` from an `IfcPropertySet`

Figure A.6 and figure A.8 (in the appendix) depict both workflows like they have been implemented in the IFC prototype utilizing the data model. Counting the necessary elements results in the following values:

$$WC(A_{\text{IFC}}) = \frac{1}{8} = 0.13$$
$$WC(B_{\text{IFC}}) = \frac{3}{7} = 0.43$$

Combining these values yields the following value:

$$\underline{WC(\text{IFC})} = \frac{(WC(A_{\text{IFC}}) + WC(B_{\text{IFC}}))}{\#Scenarios} = \frac{(0.13 + 0.43)}{2} = \underline{0.28}$$

Finally, applying the same rules to figure A.7 and figure A.9 (in the appendix), which depict the same workflows while working with IFC-R, results in the following values:

$$WC(A_{\text{IFC-R}}) = \frac{3}{10} = 0.30$$
$$WC(B_{\text{IFC-R}}) = \frac{7}{12} = 0.58$$

Resulting in:

$$\underline{WC(\text{IFC-R})} = \frac{(WC(A_{\text{IFC-R}}) + WC(B_{\text{IFC-R}}))}{\#Scenarios} = \frac{(0.30 + 0.58)}{2} = \underline{0.44}$$

Altogether, a brief summary of the additional metrics can be found in table 4.3. The values acquired in this section reveal that there are no significant differences in both data models so far. This coincides with the first metrics applied in section 4.1. However, this result only holds for the metrics used in this thesis, which will be further elaborated on in section 4.3. Furthermore, both data models have advantages and disadvantages that cannot be measured easily, which will be discussed in the next section.

Table 4.3.: Summary of the gathered values for the supporting metrics

Metric	IFC	IFC-R
Coupling Between Object Classes (CBO)	7	8
Runtime Efficiency (RE)	13	11
Data Model Understandability (DMU)	0.4	0.6
Control Flow Complexity (CFC)	0.56	0.29
Workflow Complexity (WC)	0.28	0.44

4.2.2. Applying the UCP method

So far, the application of additional metrics showed no significant differences. However, the process of acquiring the values for these metrics helps comprehend the underlying concepts of both data models. By comparing these concepts, benefits and flaws of IFC and IFC-R can be identified, which will be subject of this section with respect to the UCP method.

Due to the simplification of the value for UUCP, the application of the UCP method in that case concentrates on evaluating the Technical Complexity Factors (TCF) and Environmental Factors (EF). Hence, the aspects for this evaluation, as described in section 4.1.3, are:

- Efficiency
- Understandability
- Maintainability
- Extensibility
- Experience

Each of them will be rated on a scale from **zero** – large demand for enhancements regarding the specific use cases to **five** – no demand for further enhancements.

Efficiency First of all, the efficiency of working with the data models from the perspective of modelers and developers will be rated. In support of that, the CBO and RE metrics have been acquired, which revealed structural problems of IFC. In my opinion, there are too many "unnecessary" classes. As mentioned while retrieving the values for RE, each `IfcTypeObject` and `IfcPropertySet` will be added indirectly to an `IfcObject` by means of objectified relationships (`IfcRelDefines`). Exaggerating this issue, having 100 property sets results in an additional 100 relationship objects during runtime and in the model itself.

Furthermore, reading the properties of an object involves iterating over the relationships (via `IsDefinedBy` of an object) and getting the corresponding property set of this relationship, whereas IFC-R tries to overcome these issues by flattening the hierarchies if possible. The definition of objects involves one compartment (`ObjectDefinition`) which holds all necessary roles further describing the object. In addition to that, the design of IFC-R is strongly tied to IFC for compatibility reasons, which means leveraging that fact could improve the efficiency even more. Moreover, additional roles can be implemented easily, which will be part of the discussion in the next paragraph. In the end, it will receive a rating of **2** for IFC and **4** for IFC-R.

Understandability Working efficiently with a (meta)model goes along with a good comprehension of its important aspects and concepts. In order to make the understandability perceivable, the metrics DMU and CFC have been acquired. As mentioned above, the definition of objects with the help of objectified relationships is inconvenient. Resolving these relationships results in a less complex control flow in IFC-R as it has been measured for the CFC. Furthermore, the concept of typed objects and further definitions by means of properties seems too generic, which leads to the workarounds described in chapter 1 and 2.

IFC-R employs the concept of roles, which might be easier to grasp for modelers. In fact, the semantic meaning of roles is stronger than the concept of simple properties further describing objects, and they bring along additional advantages during the runtime, which is helpful for developers. On top of that, IFC-R is able to implement roles beyond the scope of IFC enabling the modelers to increase the semantic meaning of their models even further.

However, as described while acquiring the DMU metric, the developed transformation workflow introduces new pitfalls with respect to the overall complexity. In consequence, IFC scores with 2 again and IFC-R reaches a rating of 3, due to the relatively new concept of role-oriented modeling and the additional overhead.

Maintainability Comparing the values for the CBO metric for both data models did not reveal any differences. However, the large hierarchies and unnecessary relationships hamper the maintainability of models and the standard itself. I believe, this issue can and should be tackled for further iterations of IFC. In addition to that, shorter release cycles and more sophisticated methods for fixing issues related to shortcomings should be followed up on. Modelers and developers should be able to address certain issues without the need of workarounds or, otherwise, waiting several years.

As described in chapter 3, IFC-R can be considered an additional layer or a decorator of IFC in its current state. This especially enables developers to fix shortcomings in IFC, for example by applying additional steps in the transformation workflow, in order to improve the maintainability. However, although necessary, keeping the CROM and CROI consistent at all times decreases the maintainability by introducing further complexity, which is reflected by the values for WC. To be precise, having a strict metamodel like CROM is useful to prevent the application of workarounds such as in IFC, yet, to the disadvantage of maintainability.

As a result, IFC will be rated with a 1 because the necessity to develop own solutions, altering models by hand, or wait years for a new official version fixing an issue is not contemporary. In contrast, IFC-R has several hooks to address the mentioned issues without waiting for a new release of IFC. The basis of IFC-R, namely CROM, can be used to prevent workarounds, nevertheless, it introduces another level of complexity resulting in a score of 3.

Extensibility Maintainability is often associated with extensibility. In this context, being able to extent IFC or IFC-R means adding more information, more semantic meaning, and more value to the models in general. With respect to the overall idea of BIM, that means for example differentiating between a normal door and a fire door at a glance. However, iterating over all relationships of an `IfcObject` in order to find all property sets or the `IfcTypeObject` defining the type of it is cumbersome. This is a general problem associated with the structure of IFC.

In contrast, identifying and highlighting such differences is one of the core advantages of role-oriented modeling, which contributes to fulfilling the needs of BIM. Roles, as the main driver of IFC-R, introduce a dynamic view onto the associated objects increasing the semantic meaning. In addition to that, they can be constrained in order to prevent misuse or misconceptions. That means, having a role `FireDoor` being played by an `IfcDoor` identifies that door unambiguously as a fire door without increasing the depth of the inheritance hierarchy, which obviously is important in IFC. Therefore, evaluating the extensibility of both data models results in a score of 2 for IFC and a 4 for IFC-R.

Experience Lastly, the environmental factor "experience" sets the focus on the actors employing the data models. This factor will be evaluated from the viewpoint of modelers and developers already working with IFC because learning either of the data models from scratch is equally hard, in my opinion. Furthermore, IFC-R in its current state sits on top of IFC, which is why learning and understanding the basics of IFC is necessary anyways. As described in section 4.1.4, this factor is usually evaluated by interviews or by analyzing the work of people utilizing the data models for given tasks. Since that exceeds this work, this factor will be simply assessed by estimating the learning curve for IFC-R with respect to IFC.

For example, if a modeler is familiar with the concept of typing with `IfcTypeObjects` and further definitions of objects with `IfcPropertySets` than the transition to IFC-R is small. The reason for that, is a strong orientation on IFC during the development of IFC-R. Adding a property set with the help of a relationship to an object or adding the same set as a role to the same object should not be difficult, in my opinion. This applies to all concepts adapted in IFC-R, which is why IFC-R scores with a **3** compared to IFC scoring with a **4**.

After the evaluation of all factors, the final values for TCF and EF can be calculated. In contrast to Karner, the authors of [ONK11] did not mention any weights for their adapted factors. Therefore, all factors will be included without additional weights except **Maintainability** and **Extensibility** because they reflect the main goals of this thesis. That means, they will be rated with a weight of two. Under this premise, the results for TCF and EF for IFC are as follows:

$$TCF(\text{IFC}) = 0.6 + (0.01 \times (2 + 2 + 1 \times 2 + 2 \times 2)) = 0.7$$
$$EF(\text{IFC}) = 1.4 + (-0.03 \times 4) = 1.28$$

Resulting in a final UCP value of:

$$\underline{UCP(\text{IFC})} = 1 \times TCF(\text{IFC}) \times EF(\text{IFC}) = 1 \times 0.7 \times 1.28 = \underline{0.896}$$

Applying the same equations to IFC-R results in the following:

$$TCF(\text{IFC-R}) = 0.6 + (0.01 \times (4 + 3 + 3 \times 2 + 4 \times 2)) = 0.81$$
$$EF(\text{IFC-R}) = 1.4 + (-0.03 \times 3) = 1.31$$

Finally, resulting in:

$$\underline{UCP(\text{IFC-R})} = 1 \times TCF(\text{IFC-R}) \times EF(\text{IFC-R}) = 1 \times 0.81 \times 1.31 = \underline{1.061}$$

It is worth mentioning again that the original UCP method by Karner is a technique for estimating the initial project size. That means, the higher this value is the larger is the scope of a project. The method, though, has been adapted in a way that the higher the UCP value is the better a data model accomplishes the mentioned use cases. As a result, it can be said that IFC-R does accomplish the examined use cases and factors better than IFC.

Table 4.4.: Summary of the evaluation of TCF, EF and UCP

	IFC	IFC-R
Efficiency	2	4
Understandability	2	3
Maintainability	1	3
Extensibility	2	4
Experience	4	3
Technical Complexity Factors (TCF)	0.7	0.81
Environmental Factors (EF)	1.28	1.31
Use Case Points (UCP)	0.896	1.061

Nevertheless, the difference between IFC and IFC-R (see table 4.4) according to the applied metrology is insignificant. Possible reasons for that along with potential errors, limitations and optimizations of the evaluation will be discussed in the upcoming section.

4.3. Problems and Interim Conclusion

The second research question of this thesis is: "(RQ2) Is role-oriented modeling a solution for the identified deficiencies of IFC?" In order to answer that question, a role-oriented approach for IFC has been developed and evaluated against the existing standard. The results, however, revealed no significant differences with respect to the applied metrology. Since this does not automatically mean that IFC-R is not an improvement of IFC, this section will discuss potential errors of the analysis, elaborate on limitations, and interpose optimizations. This then leads to a more deliberate answer to the research question.

Potential Errors Before discussing possible errors, it should be mentioned that sources of errors have been prevented to the best of my belief. However, due to an insufficient consensus of measurement methods in computer science and software engineering, mistakes regarding the metrology cannot be ruled out and have to be addressed.

First of all, the measurement method itself could be an inaccurate choice or the adaptations that were made could be inappropriate for measuring and evaluating the intended goals. The intent of the UCP method is the estimation of the project size in order to calculate the needed resources in an early development stage. Its focus on technical and environmental factors appeared suitable, in my opinion. Known issues regarding the subjectivity of the method have been reduced by supporting metrics, which fit well with respect to perceiving the complexity of the data models, as far as I am concerned.

Secondly, this thesis comprises the development of IFC-R. Although the analysis of IFC is a main goal, analyzing the standard in depth and comprehending every peculiarity exceeds the possibilities of this work. In consequence, the knowledge and comprehension about the underlying concepts of IFC-R is more graspable to me compared to IFC, which could have potentially led to false assessments during the application of the UCP method.

Lastly, with regard to the upcoming elaboration on limitations, the selected use cases and scenarios for several metrics do not cover every aspect of IFC or IFC-R. Additionally, the chosen examples could be not representative enough for certain metrics or simply to small in order to be representative. The main reason for these restrictions is the fact that a comprehensive evaluation of the data models could be a separate topic for a thesis itself.

Limitations As mentioned above, this thesis is subject to several limitations. To be more precise, e.g. metrics like the Data Model Understandability (DMU) should be evaluated with the help of interviews or questionnaires according to [ISO15]. However, this was not possible during this thesis diminishing the acquired values for that metric. Additionally, regarding the extent of this work, Industry Foundation Classes is a huge standard that has numerous classes, property sets, type objects, and core concepts. That is why IFC-R only focuses on improving the found issues regarding `IfcPropertySets` and `IfcTypeObjects` (see chapter 2). This mean in particular that the evaluation covers only a small part of IFC, hence, raising no claim to completeness.

Another premise of the evaluation is the high degree of subjectivity of the Use Case Points method. Despite trying to reduce this circumstance by utilizing supporting metrics, in its essence UCP remains a rather subjective measurement method. Furthermore, some of the additional metrics are self-provided. That means that their validity, i.e. do they measure what they are intended to measure, needs an evaluation as well. All things considered, most limitations result from the lack of available measurement methods for comparing two data models or modeling languages by means of numbers.

Optimizations The limitations and the evaluation in general have several starting points for optimizations with respect to future work.

Firstly, an extensive empirical evaluation of IFC could be a good start for future work in order to comprehend the structure and concepts by means of numbers. Quantifying the found deficiencies could enhance the development process of an improved approach and would accelerate the comparison because the metrology and the metrics would be known beforehand. However, other suitable measurement methods need to be identified, and the new solution should not be tied too strongly to the evaluation – otherwise it will be biased.

Secondly, as mentioned above, the setups, use cases, and examples used to acquire the values for the metrics are rather small. Measurement such as the Workflow Complexity are more convincing when considering more different workflows. However, I am of the opinion that the most representative setups have been used. Yet, it is possible that other important use cases are missing, which diminishes the results.

Lastly, most of the metrics could be further tuned in order to achieve more precise results. As mentioned above, DMU should be retrieved through interviews because it currently reflects the view of the author of this thesis only. Additionally, the CBO metric is intended for the evaluation of object-oriented applications, which might be unsuitable for a role-oriented approach. On top of that, the CFC and WC metrics are pretty similar to the Cyclomatic Complexity which was already excluded at the beginning of the evaluation. Furthermore, the weighting of the Technical Complexity Factors (TCF) and Environmental Factors (EF) should be further adjusted, or single factors could be exchanged for more appropriate aspects.

Altogether, the evaluation is not free of flaws, which is why the next paragraph will draw an interim conclusion with respect to the acquired values. On the one hand, IFC-R performs better in most of the supporting metrics and in the UCP method. With respect to the examined use cases, that means that IFC-R does improve the intended issues.

On the other hand, the measured differences are fairly small. The assumption to be made here is that the problems of IFC, as described in chapter 2, can be improved by enhancing the object-oriented aspects of the Industry Foundation Classes. More precisely, removing most of the deprecated concepts and misconceptions regarding object orientation would also improve IFC in general, in my opinion.

As a result, RQ2 can be answered with **Yes** because role-oriented modeling can be considered a more mature object-oriented approach. Beyond that, role-oriented modeling and programming has further benefits such as enhanced handling during runtime. The evaluation shows that IFC-R can take advantage of these benefits without any loss in quality. Especially applications for later stages in the BIM workflow, e.g. an application for the facility management of a building, can implement IFC-R to utilize the abovementioned improvements regarding IFC and the advantages of roles.

To put it in a nutshell, the development of IFC-R does not resolve every issue, but it – along with the comparison of both approaches – contributes to a deeper comprehension and sets a solid basis for further research in order to improve Industry Foundation Classes.

5. Conclusion and Outlook

The goal of this thesis was to improve the extensibility and maintainability of Industry Foundation Classes. Recent research in the area of model-driven software engineering had identified weaknesses with respect to the expressiveness of the semantic meaning of the models and the modeling concepts of IFC in general.

Inevitably, the named problems lead to workarounds in the community to overcome these deficiencies, which in turn results in further issues. A standard like IFC should not have such issues because it is intended to be used as a solid basis for the overall idea of BIM. That explains the severe demand for improvements.

In order to achieve this task, it was crucial to understand IFC with its main ideas and concepts. More importantly, this analysis enabled a profound comprehension of the issues of this modeling language and the identification of aspects for improvements, which was the first research question of this thesis:

(RQ1) What are the (core) issues of IFC and how can they be improved?

With regard to this question, an extensive analysis of the documentation, the underlying modeling language EXPRESS, well-known tools for working with IFC, and real world examples resulted in the identification of the two core issues that were subject to this thesis. The adding of properties by means of **property sets** and the orthogonal classification with the help of **object typing** diminish the expressiveness of the resulting models and the overall informative content. The entire analysis has been translated into a concise overview that serves as the basis for the intended improvements.

The first main contribution of this thesis – the identification and deep analysis of the problems – showed that IFC employs certain object-oriented concepts. However, some of them are not well utilized and others have been applied in an overcomplicated way, which results in an improper object orientation. Fortunately, impulses for more sophisticated approaches that might help improve the extensibility and maintainability of IFC were identified by the MDSE community.

One of them being **role-oriented modeling** has been selected due to its easy-to-understand nature, its state-of-the-art concepts, and its advantages regarding the runtime features of applications. This selection has lead to the second research question:

(RQ2) Is role-oriented modeling a solution for the identified deficiencies of IFC?

In order to answer this question, a novel approach called Industry Foundation Classes with Roles (IFC-R) was developed. The main idea was to capture the dynamic and context-dependent characteristics of property sets and object types with the help of roles. The outcome of this development is a workflow that stepwise transforms IFC models into a role-oriented representation of the input model with the help of Compartment Role Object Models (CROM).

Nevertheless, representing IFC models with roles is not automatically an answer to the second research question. First, it is necessary to show that a role-oriented approach like IFC-R improves the extensibility and maintainability of IFC. As a second step, IFC-R should be usable in the same way as IFC without any loss in quality. Consequently, a comparison of both IFC and IFC-R has been conducted as a proof for the former, and prototypes have been developed to show the latter.

The developed prototypes demonstrated that both data models can be used analogously in an application without any shortcomings. Yet, the use cases and the prototypes themselves are not complex enough to sell the real advantages of roles during runtime convincingly. In addition, the comparison of IFC and IFC-R revealed no significant enhancements with respect to the applied measurement method. Despite the fact that the measured values did not show a significant dominance of IFC-R compared to IFC, the developed solution and the evaluation itself still form an important contribution to the research on the improvement of IFC.

Due to the nature and the time frame of a thesis like this, simplifications and open issues were inevitable. As mentioned before, the developed prototypes are not able to reflect the true advantages of role-oriented programming because the use cases are fairly small for this task. For example, measuring the runtime environment of two applications for an appropriate comparison only makes sense for software with a proper size.

Speaking of measurements, – compared to other sciences – computer science struggles with the lack of standardized measurement methods, which has already been discussed extensively in section 4.3. This might have influenced the evaluation as well, although the methodology was chosen to the best of my knowledge and belief.

Furthermore, the approach to the role-oriented version of IFC that was developed is only one way of addressing the idea. It was strongly tied to IFC in order to be compatible and available for back transformation. If IFC-R were less strongly tied to IFC, the capabilities of such a role-oriented approach might have improved the intended aspects even further, yet this remains subject to further examination.

This being said, more potential for further research can be identified. The basic analysis of IFC's core concepts has been acquired without having a particular solution in mind, which means, it can serve as a basis for other approaches as well. For example, a multi-level modeling approach such as *Deep Instantiation* can also start by elaborating on the issues of property sets and object typing. This different aim could possibly lead to new conclusions.

Secondly, IFC-R in its current state only represents the necessary parts of IFC. Consequently, it should be further developed in order to create real world applications for the industry. An application of such size would then be able to reflect the above-mentioned advantages of role-oriented programming properly.

This automatically opens up a third perspective for future work as the entire applications could then be compared extensively with each other. This would support the comparison results of this thesis because such an evaluation could rely on standardized software measurement methods. Thus, the advantages of IFC-R could be demonstrated more validly in order to support the hypothesis that role-oriented modeling is a reasonable solution for the identified deficiencies of IFC.

All things considered, any further research in this area is crucial because the digitalization of the building industry has just begun and it might be fatal to advance this evolution with an insufficient standard like IFC.

References

[Abr10] Alain Abran. *Software Metrics and Software Metrology*. IEEE Computer Society, Wiley, 2010. ISBN: 0470606827.

[Ais86] Robert Aish. "Building modelling: the key to integrated construction CAD". In: *CIB 5th International Symposium on the Use of Computers for Environmental Engineering Related to Buildings*. Vol. 5. 1986, pp. 7–9.

[AK01] Colin Atkinson and Thomas Kühne. "The Essence of Multilevel Metamodeling". In: *International Conference on the Unified Modeling Language*. Springer. 2001, pp. 19–33.

[Ara20] Vincent Aranega. *PyEcore - GitHub Repository*. June 2020. URL: `https://github.com/pyecore/pyecore` (visited on 09/26/2020).

[Bor+15a] Stefano Borgo, Emilio M. Sanfilippo, Aleksandra Šojić, and Walter Terkaj. "Ontological Analysis and Engineering Standards: an initial study of IFC". In: *Ontology Modeling in Physical Asset Integrity Management*. Springer, 2015, pp. 17–43.

[Bor+15b] André Borrmann, Markus König, Christian Koch, and Jakob Beetz. *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. Springer-Verlag, 2015.

[bui18a] buildingSMART. *Industry Foundation Classes Version 4.1.0.0*. June 2018. URL: `https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/` (visited on 05/21/2020).

[bui18b] buildingSMART. *Industry Foundation Classes Version 4.1.0.0 - Foreword*. June 2018. URL: `https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/link/foreword.htm` (visited on 08/02/2020).

[bui18c] buildingSMART. *Industry Foundation Classes Version 4.1.0.0 - Introduction*. June 2018. URL: `https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/link/introduction.htm` (visited on 08/02/2020).

[bui20] buildingSMART. *buildingSMART International - Webpage*. 2020. URL: `https://www.buildingsmart.org/` (visited on 08/02/2020).

[BVD09] Jakob Beetz, Jos Van Leeuwen, and Bauke De Vries. "IfcOWL: A case of transforming EXPRESS schemas into ontologies". In: *Ai Edam* 23.1 (2009), pp. 89–101.

[CAG16] Victorio Albani Carvalho, João Paulo A. Almeida, and Giancarlo Guizzardi. "Using a Well-Founded Multi-level Theory to Support the Analysis and Representation of the Powertype Pattern in Conceptual Modeling". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2016, pp. 309–324.

[Coe20] Aurélien Coet. *StatiCFG - GitHub Repository*. June 2020. URL: https://github.com/coetaur0/staticfg (visited on 07/20/2020).

[Eas+11] Chuck Eastman, Paul Teicholz, Rafael Sacks, and Kathleen Liston. *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*. John Wiley & Sons, 2011.

[Ecl20] Eclipse Foundation Inc. *Eclipse Epsilon - Documentation*. Sept. 2020. URL: https://www.eclipse.org/epsilon/ (visited on 09/26/2020).

[FRN15] Tarcisio Mendes de Farias, Ana Roxin, and Christophe Nicolle. "IfcWoD, semantically adapting IFC model relations into OWL properties". In: *arXiv* (2015).

[Gam+94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[Göt+19] Sebastian Götz, Andreas Fehn, Frank Rohde, and Thomas Kühn. "Model-driven Software Engineering for Construction Engineering: Quo Vadis?" In: *Journal of Object Technology* (2019).

[Hal77] Maurice Howard Halstead. *Elements of software science*. Vol. 7. Elsevier New York, 1977.

[HM08] Terry Halpin and Tony Morgan. *Information Modeling and Relational Database*. Elsevier Morgan Kaufmann, 2008.

[Hoo18a] Emma Hooper. *FC (Industry Foundation Classes) – An introduction for Autodesk Revit users*. Sept. 2018. URL: https://bimblog.bondbryan.co.uk/ifc-industry-foundation-classes-an-introduction-for-autodesk-revit-users (visited on 08/22/2020).

[Hoo18b] Emma Hooper. *IFC (Industry Foundation Classes) – Predefined Types in Autodesk Revit*. Nov. 2018. URL: https://bimblog.bondbryan.co.uk/ifc-industry-foundation-classes-predefined-types-in-autodesk-revit/ (visited on 08/23/2020).

[Hoo19] Emma Hooper. *IFC (Industry Foundation Classes) – IFC Properties in Autodesk Revit*. Mar. 2019. URL: https://bimblog.bondbryan.co.uk/ifc-industry-foundation-classes-ifc-properties-in-autodesk-revit/ (visited on 05/22/2020).

[ISO04] ISO/TC 184 Automation systems and integration. *ISO 10303-11 Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual*. Nov. 2004.

[ISO13] ISO/TC 184/SC 4 Industrial data. *ISO/TR 10303-12:1997*. Dec. 2013. URL: https://www.iso.org/standard/25071.html (visited on 08/16/2020).

[ISO15] ISO/IEC JTC 1/SC 7 Software and Systems Engineering. *ISO/IEC 25024 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of data quality*. Oct. 2015.

[ISO16] ISO/TC 184 Automation systems and integration. *ISO 10303-21 Industrial automation systems and integration - Product data representation and exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure*. Mar. 2016.

[ISO18] ISO/TC 59 Building construction. *ISO 16739-1 Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries - Part 1: Data schema*. Nov. 2018.

[Kar93] Gustav Karner. "Metrics for objectory". PhD thesis. Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344: 21, 1993.

[Kri20] Thomas Krijnen. *IfcOpenShell - GitHub Repository*. Sept. 2020. URL: `https://github.com/IfcOpenShell/IfcOpenShell` (visited on 09/26/2020).

[Küh+15] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. "A Combined Formal Model for Relational Context-Dependent Roles". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. Association for Computing Machinery, 2015, pp. 113–124.

[Küh+16] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. "FRaMED: full-fledge role modeling editor (tool demo)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM. 2016, pp. 132–136.

[Küh+19] Thomas Kühn, Christopher Werner, Hendrik Schön, Zhao Zhenxi, and Uwe Aßmann. "Contextual and Relational Role-Based Modeling Framework". In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2019, pp. 442–449.

[Küh19] Thomas Kühn. *GitHub Repository for CROM*. May 2019. URL: `https://github.com/Eden-06/CROM` (visited on 09/13/2020).

[Küh20] Thomas Kühn. *FRaMED-io - GitHub Repository*. Apr. 2020. URL: `https://github.com/Eden-06/FRaMED-io` (visited on 09/12/2020).

[Lac20] Michele Lacchia. *Radon 4.1.0 - Documentation*. Jan. 2020. URL: `https://radon.readthedocs.io/en/latest/` (visited on 06/25/2020).

[LK12] Mikael Laakso and Arto Kiviniemi. "The IFC Standard - A Review of History, Development, and Standardization". In: *ITcon* 17 (2012), pp. 134–161.

[McC76] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[Mot+16] Ali Motamedi, Mohammad Mostafa Soltani, Shayan Setayeshgar, and Amin Hammad. "Extending IFC to incorporate information of RFID tags attached to building elements". In: *Advanced Engineering Informatics* 30.1 (2016), pp. 39–53.

[Mou19] Dion Moult. *How to create better IFC files with Revit*. Mar. 2019. URL: `https://thinkmoult.com/how-to-create-better-ifc-files-with-revit.html` (visited on 05/22/2020).

[OMG16] OMG. *Meta Object Facility (MOF) Core Specification 2.5.1*. Oct. 2016.

[ONK11] Miroslaw Ochodek, J. Nawrocki, and K. Kwarciak. "Simplifying effort estimation based on Use Case Points". In: *Information and Software Technology* 53.3 (2011), pp. 200–213.

[ÖO09] Hubert Österle and Boris Otto. “A Method For Consortial Research”. In: *Available at SSRN 1425907* (2009).

[Pfl08] Shari Lawrence Pfleeger. “Software metrics: Progress after 25 years?” In: *IEEE Software* 25.6 (2008), pp. 32–34.

[RFM13] João Rio, Bruno Ferreira, and João Pedro Poças Martins. “Expansion of IFC model with structural sensors”. In: *Informes de la Construcción* (2013).

[RG98] Dirk Riehle and Thomas Gross. “Role Model Based Framework Design and Integration”. In: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1998, pp. 117–133.

[RH97] Linda H. Rosenberg and Lawrence E. Hyatt. “Software Quality Metrics for Object-Oriented Environments”. In: *Crosstalk journal* 10.4 (1997), pp. 1–6.

[RWL+96] Trygve Reenskaug, Per Wold, Odd Arild Lehne, et al. *Working with objects: the OOram software engineering method*. Manning Greenwich, 1996.

[Sne95] Harry M. Sneed. “Understanding Software Through Numbers: a Metric Based Approach to Program Comprehension”. In: *Journal of Software Maintenance: Research and Practice* 7.6 (1995), pp. 405–419.

[SW94] Douglas A. Schenck and Peter R. Wilson. *Information modeling the EXPRESS way*. New York [u.a.]: Oxford Univ. Press, 1994. ISBN: 0195087143.

[The20] The Qt Company. *Qt Creator - web page*. 2020. URL: `https://www.qt.io/product/development-tools` (visited on 10/04/2020).

[Tho20] Phil Thompson. *PyQt5 - project web page*. Sept. 2020. URL: `https://pypi.org/project/PyQt5/` (visited on 10/04/2020).

[VT92] Gilles A. Van Nederveen and Frits P. Tolman. “Modelling multiple views on buildings”. In: *Automation in Construction* 1.3 (1992), pp. 215–224.

[WLW09] Matthias Weise, Thomas Liebich, and Jeffrey Wix. “Integrating use case definitions for IFC developments”. In: *eWork and eBusiness in Architecture and Construction. London: Taylor & Francis Group* (2009), pp. 637–645.

[Zhi+11] Ma Zhiliang, Wei Zhenhua, Song Wu, and Lou Zhe. “Application and extension of the IFC standard in construction cost estimating for tendering in China”. In: *Automation in Construction* 20.2 (2011), pp. 196–204.

A. Figures

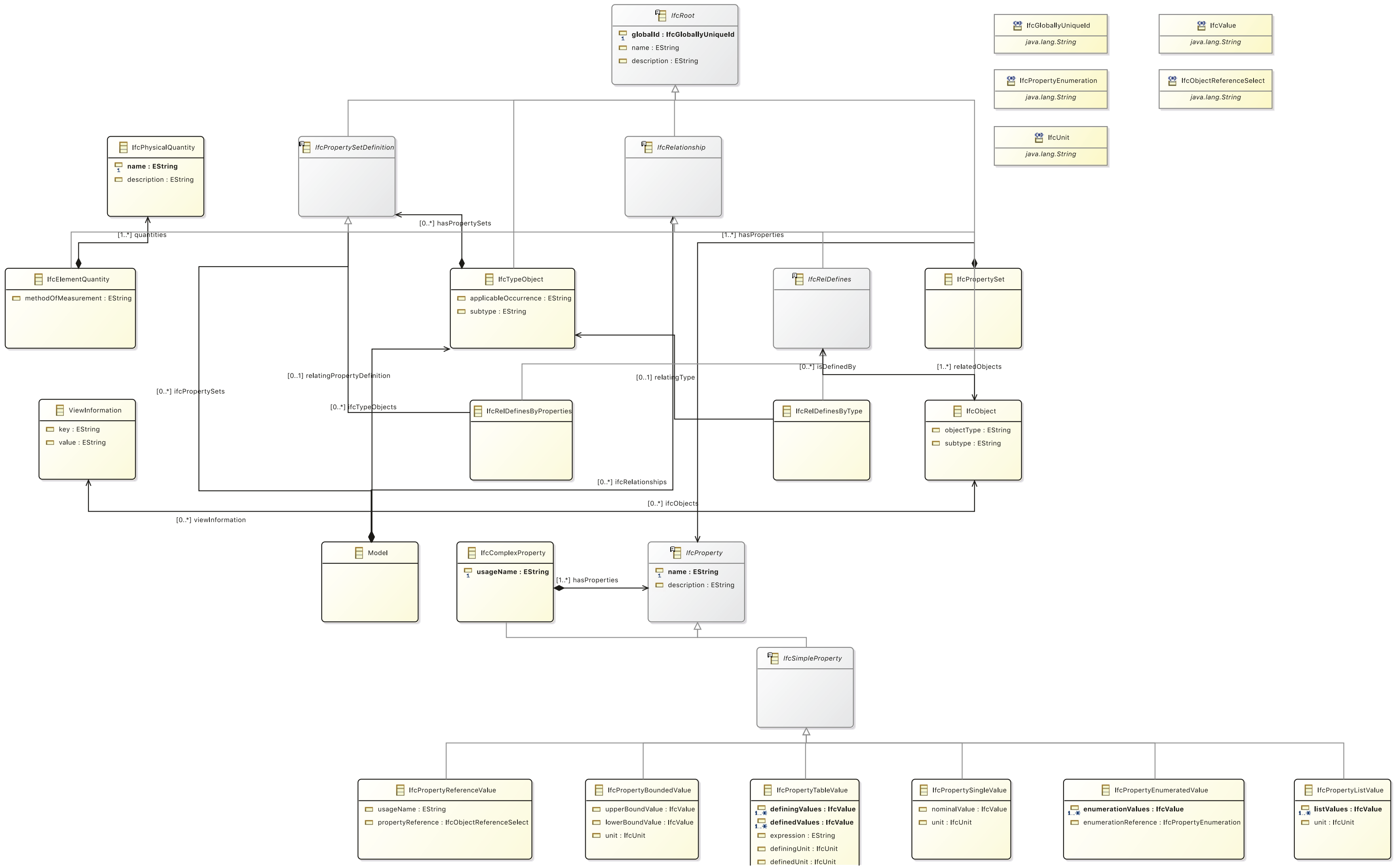


Figure A.1.: Class diagram of the IFCModel (*Ecore* metamodel)

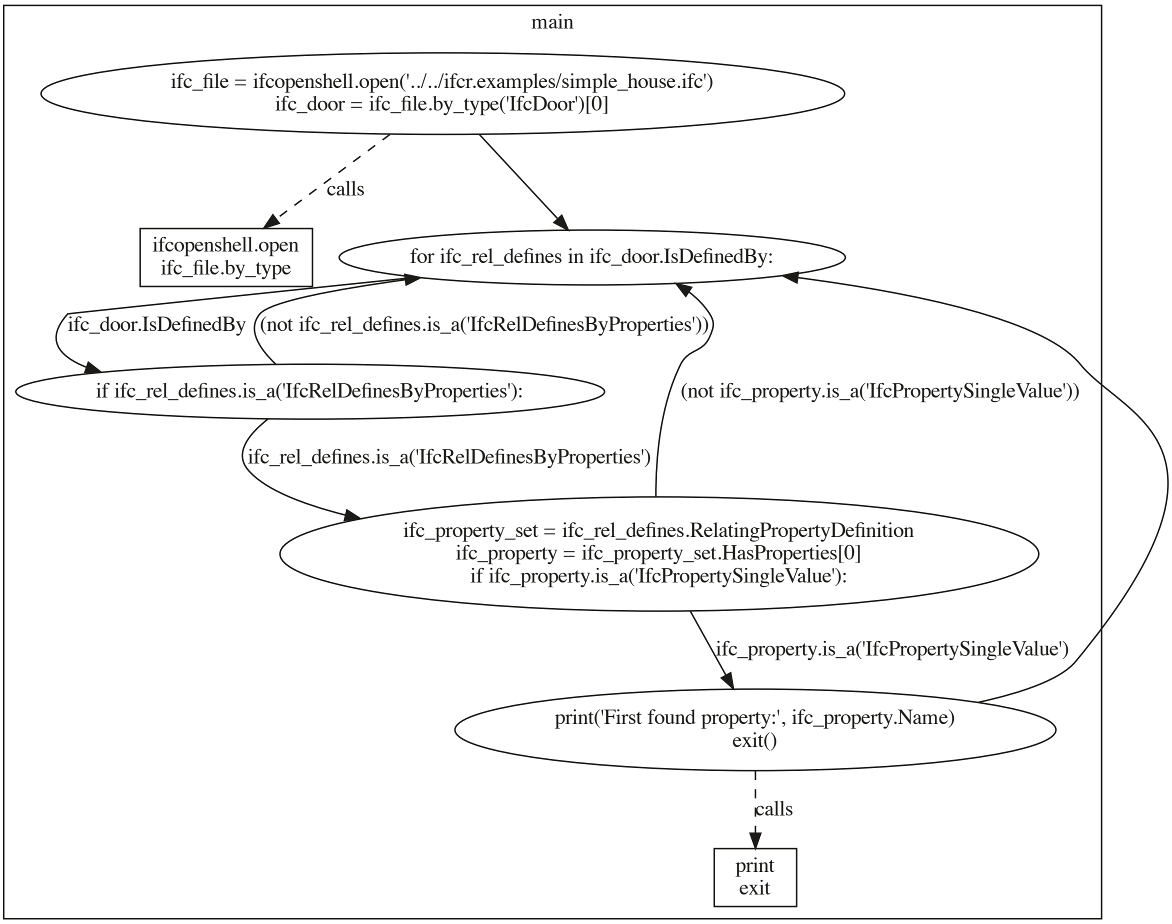


Figure A.2.: Control flow graph for the `ifc_cfc_read_property.py` script

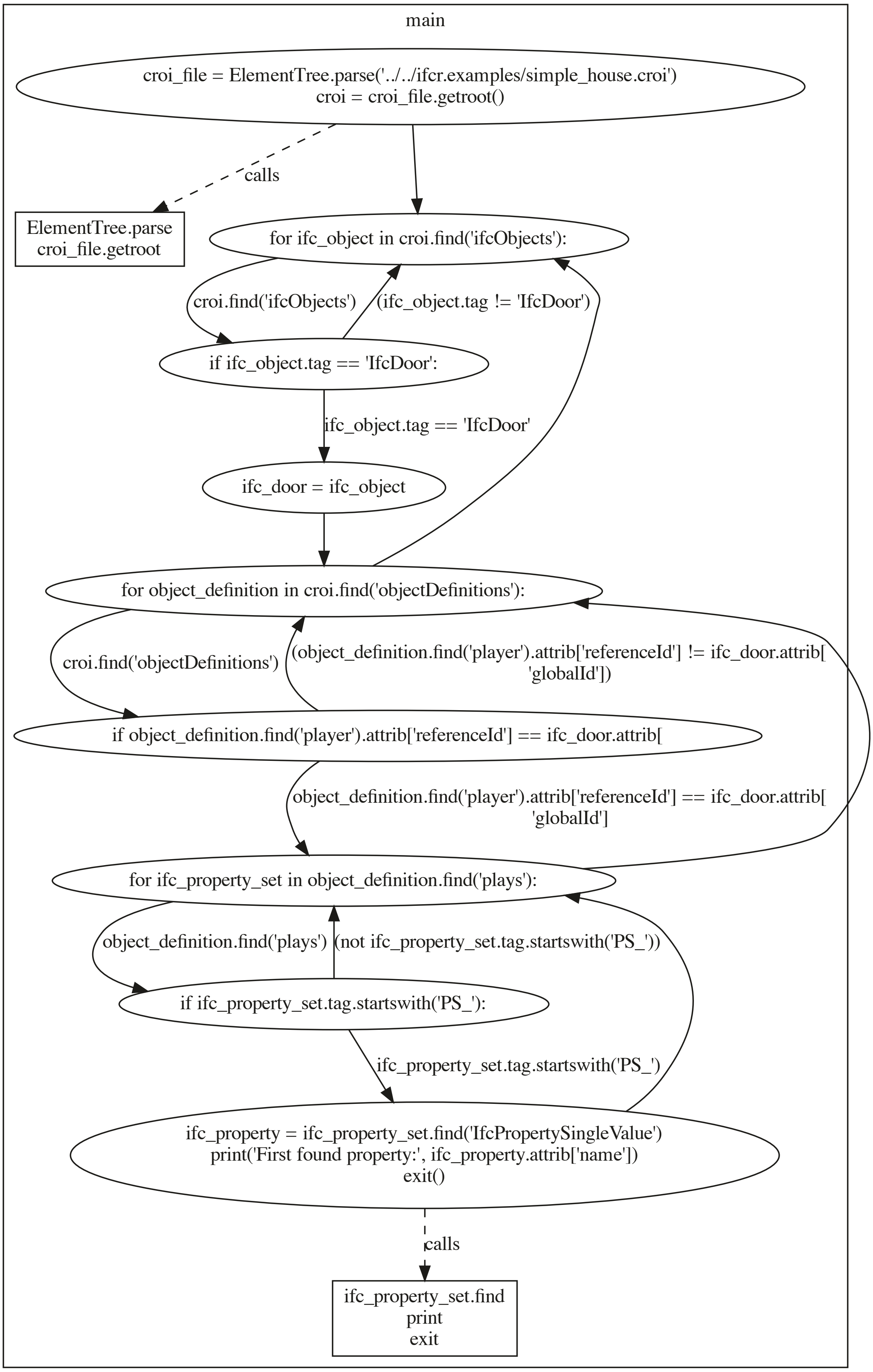


Figure A.3.: Control flow graph for the `ifcr_cfc_read_property.py` script

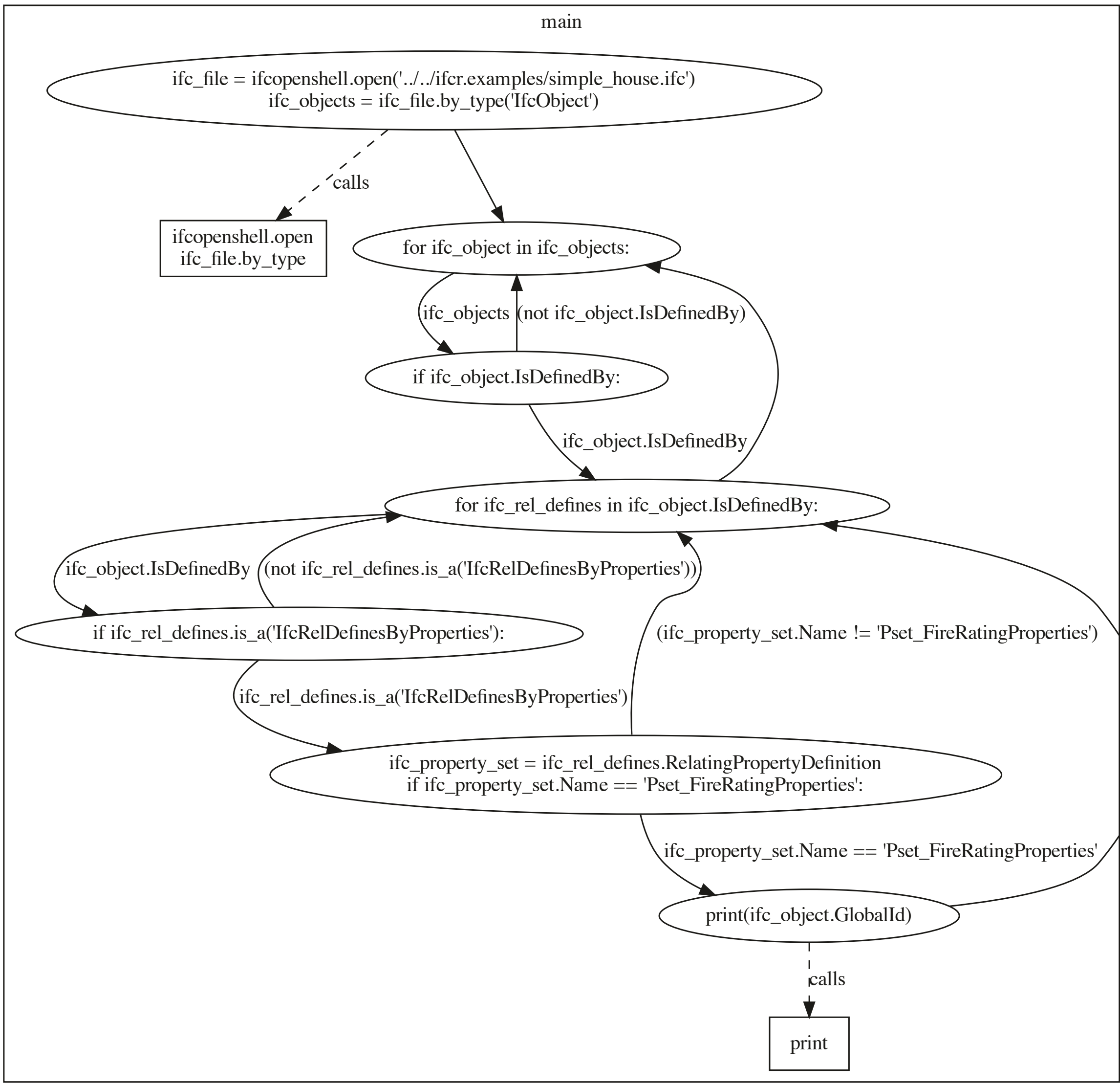


Figure A.4.: Control flow graph for the `ifc_cfc_find_objects.py` script

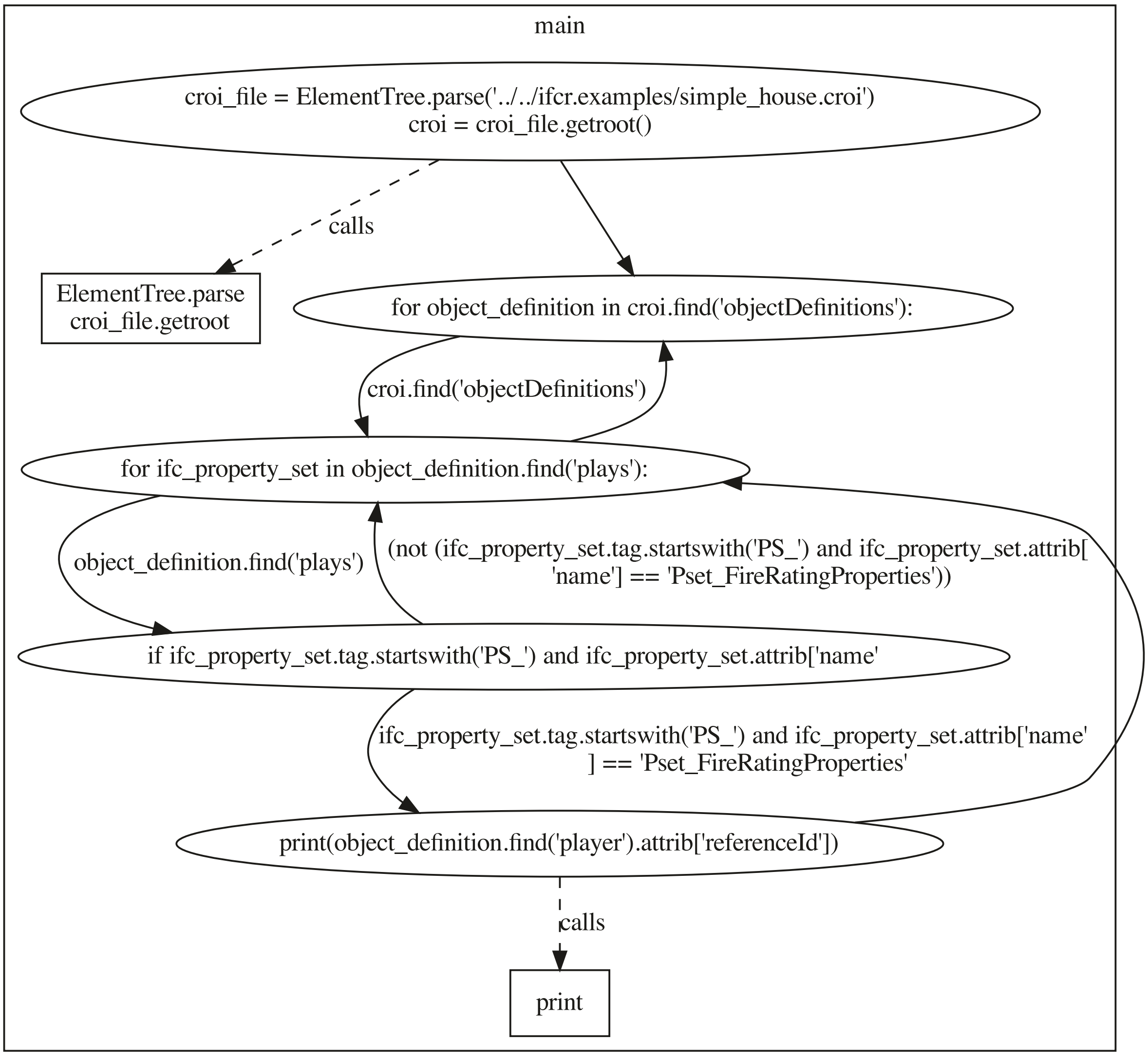


Figure A.5.: Control flow graph for the `ifcr_cfc_find_objects.py` script

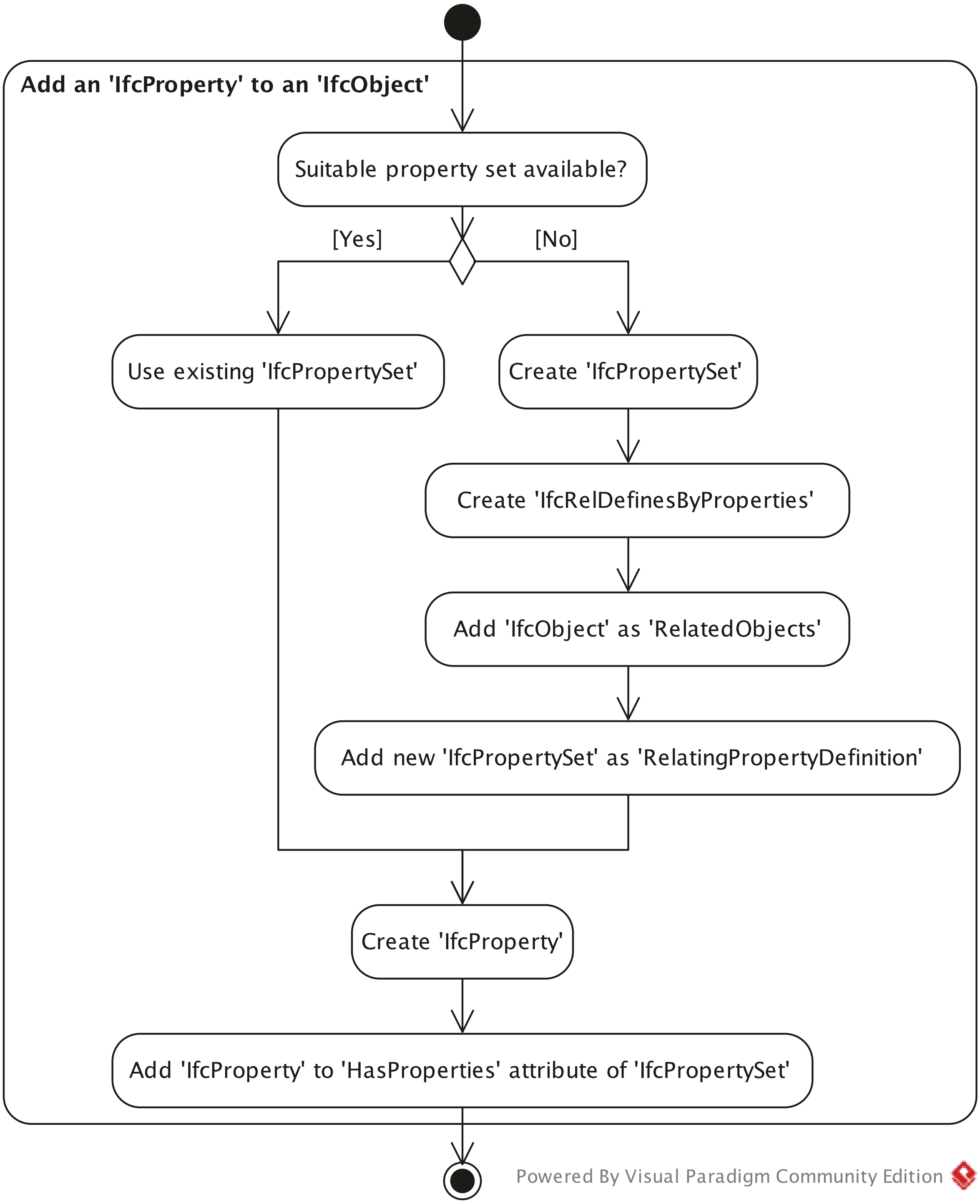


Figure A.6.: Activity diagram for adding an `IfcProperty` to an `IfcObject` in IFC

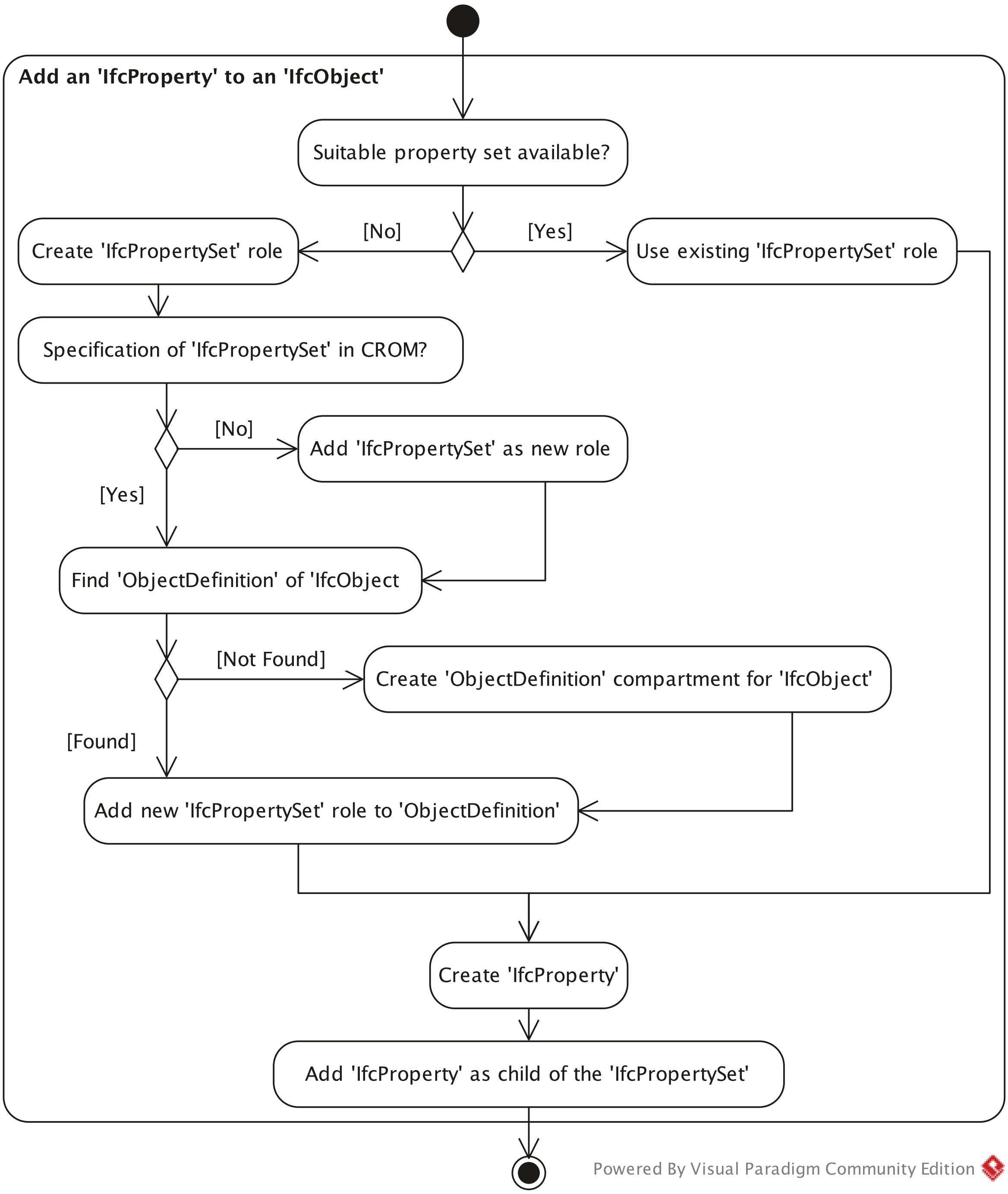


Figure A.7.: Activity diagram for adding an `IfcProperty` to an `IfcObject` in IFC-R

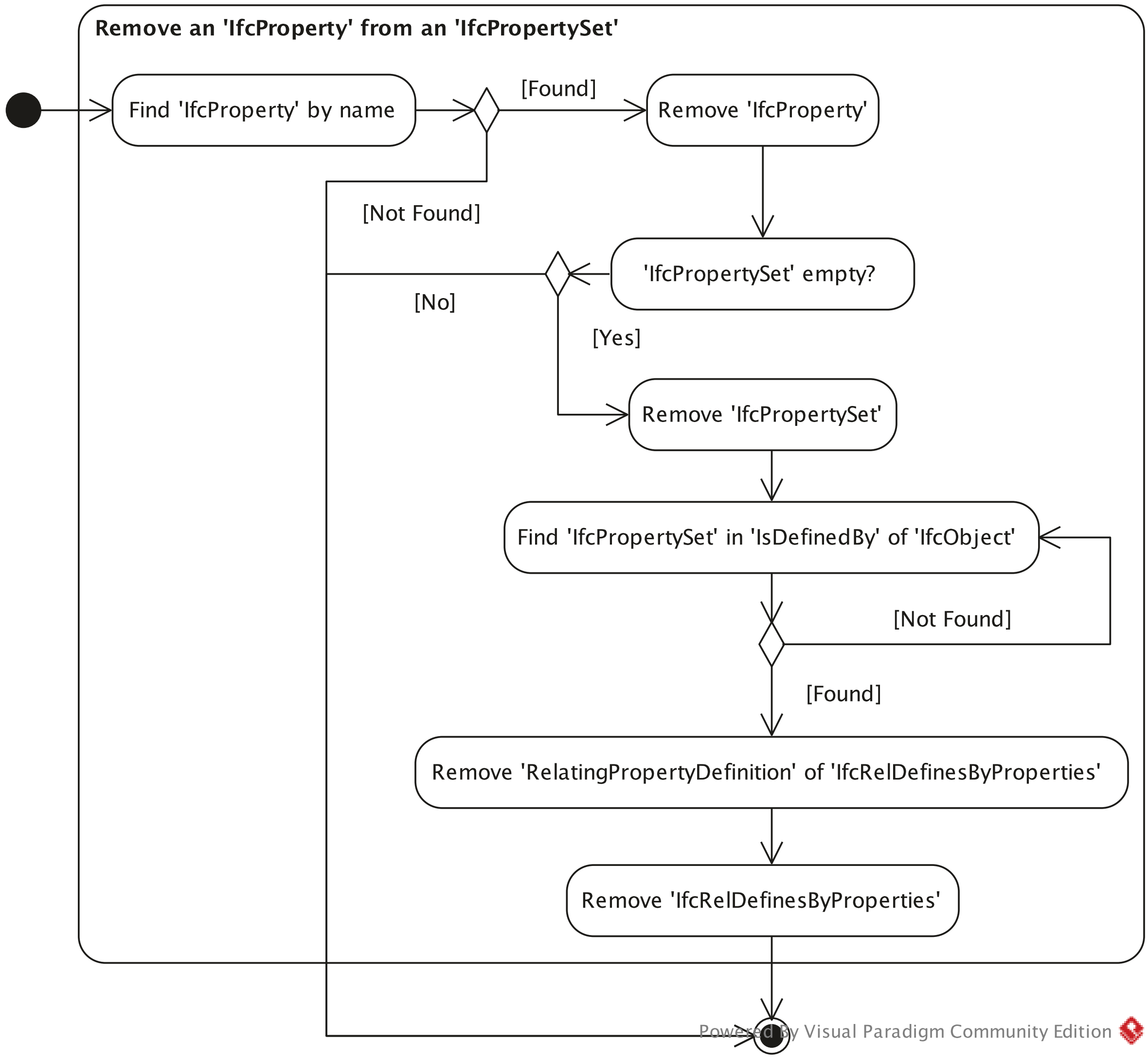


Figure A.8.: Activity diagram for removing an `IfcProperty` from an `IfcPropertySet` in IFC

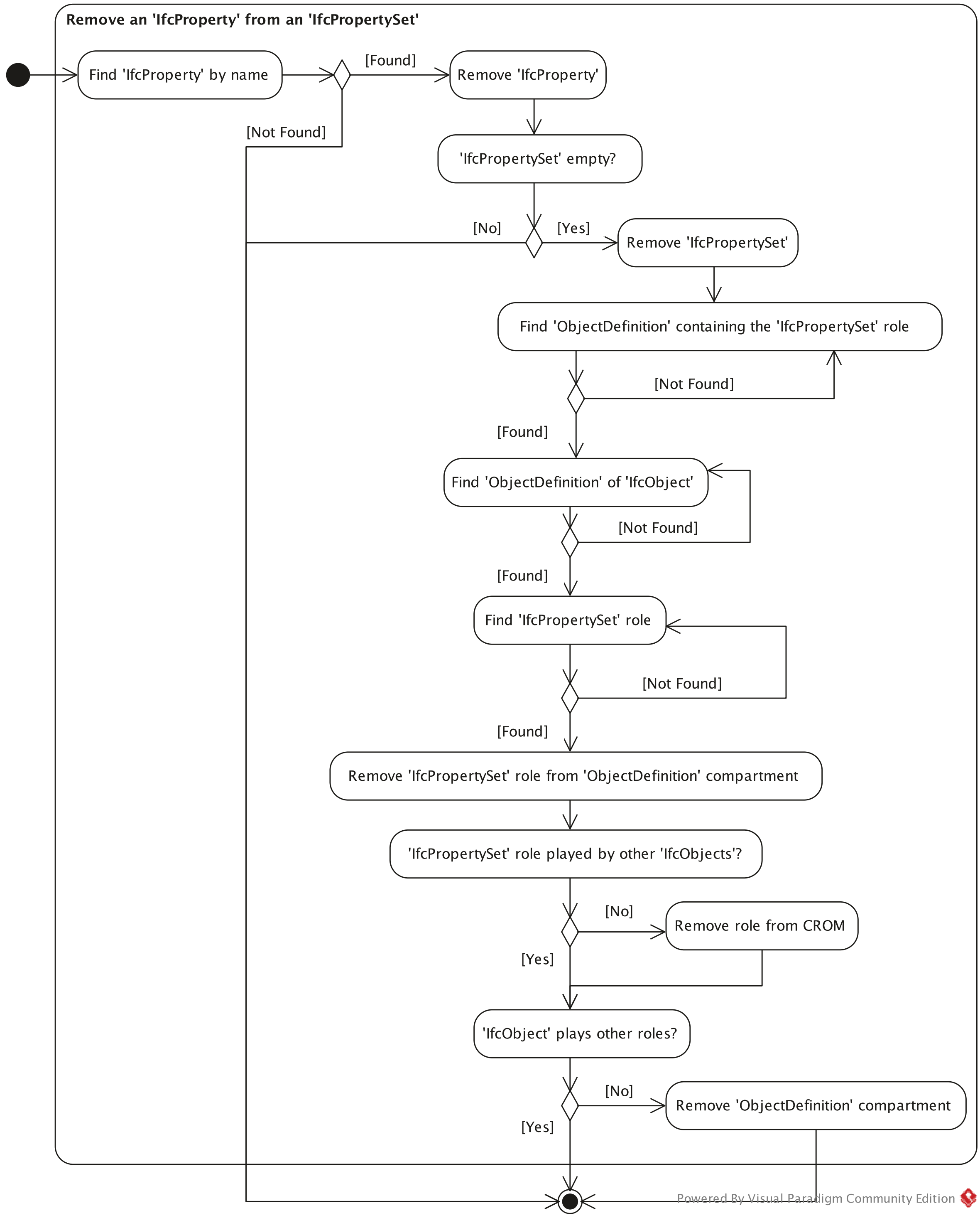


Figure A.9.: Activity diagram for removing an `IfcProperty` from an `IfcPropertySet` in IFC-R

B. Code Listings

Listing B.1: create_ifc_model.py

```python
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

"""Create an IFC model.

Proof of concept implementation for transforming IFC/STEP files into IFC models according to a given meta model.
Preprocessing step for the IFC2CROM/IFC2CROI model transformations, which enables the creation of different (domain)
views, e.g. filtering for certain 'IfcObjects'. Those views will be used by the prototypes for providing only the
necessary information for each domain.

Note: The documentation for the view mechanism can be found in the README.md.

"""

import argparse
import ifcopenshell
import json
import sys

from itertools import islice
from pathlib import Path
from pyecore.resources import ResourceSet, URI
from pyecore.utils import DynamicEPackage
from time import process_time
from views.view_extensions import *

__author__ = "Martin Klaude"
__status__ = "Prototype"


class IfcModelCreator:
    def __init__(self):
        self._ifc_objects = []
        self._ifc_filename = None
        self._ifc_file = None
        self._ifc_mm = None
```

```
        self._view_name = None
        self._view_information = {}
        self._startup()

    def create_ifc_model(self):
        """Parses the given IFC/STEP file by using the IfcOpenShell package and creates an IFC model instance with
        respect to the preloaded meta model. """
        ifc_model = self._ifc_mm.Model()
        # Iterate over the filtered view elements and handle their properties and relationships.
        total_object_count = len(self._ifc_objects)
        print(total_object_count, "IfcObject(s) will be processed in total.")
        for index, ifc_object in enumerate(self._ifc_objects):
            ifc_m_object = self._ifc_mm.IfcObject()
            self._add_ifc_root_attributes(ifc_m_object, ifc_object)
            if ifc_object.ObjectType:
                ifc_m_object.objectType = ifc_object.ObjectType
            # Add 'IfcRelationships' to the 'IfcObject'.
            for ifc_rel in ifc_object.IsDefinedBy:
                self._add_ifc_relationship(ifc_model, ifc_rel, ifc_m_object)
            ifc_model.ifcObjects.append(ifc_m_object)
            if (index + 1) % 1000 == 0:
                print(index + 1, "IfcObject(s) out of", total_object_count, "processed.")
        # Add the given view information.
        if self._view_information:
            for key in self._view_information:
                ifc_m_view_information = self._ifc_mm.ViewInformation()
                ifc_m_view_information.key = key
                ifc_m_view_information.value = self._view_information[key]
                ifc_model.viewInformation.append(ifc_m_view_information)
        self._save_ifc_model(ifc_model)

    def _startup(self):
        """Handles the startup of the script with the following steps:
                + Checks the command line arguments
                + Opens the IFC/STEP file and checking the 'FILE_SCHEMA' (currently only IFC2X3 is supported)
                + Loads the corresponding IFC meta model
        """
        arg_parser = argparse.ArgumentParser(
            description='Parses an IFC/STEP file and creates an appropriate IFC model.')
        arg_parser.add_argument('filename', metavar='file', help='A valid IFC/STEP file.')
        arg_parser.add_argument('configuration', metavar='configuration', nargs='?',
                                                          help='A configuration file (JSON) for the extensions.')
        args = arg_parser.parse_args()
        self._ifc_filename = args.filename
        self._ifc_file = ifcopenshell.open(self._ifc_filename)
        if self._ifc_file.schema != "IFC2X3":
            sys.exit("Currently only IFC2X3 is supported!")
        self._load_ifc_meta_model()
        if args.configuration:
            self._apply_view_extensions(args.configuration)
        else:
            self._ifc_objects = self._ifc_file.by_type('IfcObject')

    def _apply_view_extensions(self, extensions_config):
        """Hook method to apply the view extensions. Check the documentation for further
```

```
information.

                :param extensions_config: The configuration file (JSON) passed as command line argument.
        """
        view_extensions = ViewExtensions(self._ifc_file)
        with open(extensions_config, 'r') as config_file:
            config = json.load(config_file)
            self._view_name = config['view_name']
            for extension in islice(config, 1, None):
                extension_method = getattr(view_extensions, extension, view_extensions.method_not_found)
                extension_method_params = []
                for extension_config in config[extension]:
                    extension_method_params.append(str(config[extension][extension_config]))
                if extension.startswith('filter_'):
                    # Only extension methods starting with 'filter_' will modify the list of preprocessed
                    # 'IfcObjects'.
                    self._ifc_objects.extend(extension_method(*extension_method_params))
                elif extension.startswith('vi_'):
                    key, value = extension_method(*extension_method_params)
                    self._view_information[key] = value
                else:
                    # Simple execution of the extension method.
                    extension_method(*extension_method_params)

    def _load_ifc_meta_model(self):
        """Loads the IFC meta model by using the PyEcore package and creates all necessary modeling elements. """
        try:
            # Import the IFC meta model and register it in a ResourceSet (PyEcore).
            ifc_mm_rset = ResourceSet()
            ifc_mm_resource = ifc_mm_rset.get_resource(URI('../ifcr.metamodels/IFCModel.ecore'))
            ifc_mm_root = ifc_mm_resource.contents[0]
            ifc_mm_rset.metamodel_registry[ifc_mm_root.nsURI] = ifc_mm_root
            # Create all elements (EClasses, etc.) from the IFC meta model (Ecore).
            self._ifc_mm = DynamicEPackage(ifc_mm_root)
        except Exception as e:
            sys.exit("An error occurred while loading the IFC meta model! Error: \n\t%s" % e)

    @staticmethod
    def _add_ifc_root_attributes(ifc_m_element, ifc_element):
        """Adds common properties (due to inheritance from 'IfcRoot') to the given IFC modeling element.

                :param ifc_m_element: The IFC modeling element for adding common properties.
                :param ifc_element: The IFC element (which inherits from 'IfcRoot').
        """
        ifc_m_element.globalId = ifc_element.GlobalId
        if ifc_element.Name:
            ifc_m_element.name = ifc_element.Name
        if ifc_element.Description:
            ifc_m_element.description = ifc_element.Description
        if ifc_element.is_a('IfcObject') or ifc_element.is_a('IfcTypeObject'):
```

```python
            # Add the actual subtype of 'IfcObject' and 'IfcTypeObject' for further distinction because the name is an
            #  optional attribute in IFC.
            # Note: Simplified due to the fact of being a POC. Should be handled differently in a real implementation.
            ifc_m_element.subtype = ifc_element.is_a()

    def _add_ifc_relationship(self, ifc_model, ifc_rel, ifc_m_object):
        """Adds an 'IfcRelationship' to the given model element and handles further processing with respect to the type
        of relationship. An 'IfcRelDefinesByProperties' relationship handles the creation of property sets and refers
        them to itself and the given object. An 'IfcRelDefinesByType' relationship does the same with type objects.

                :param ifc_model: The IFC model (root) element.
                :param ifc_rel: The IFC element containing the relationship.
                :param ifc_m_object: The IFC object which will be defined by the given relationship.
        """
        ifc_m_rel = None
        # Check if relationship already exists and add attributes to existing one.
        for ifc_relationship in ifc_model.ifcRelationships:
            if ifc_relationship.globalId == ifc_rel.GlobalId:
                ifc_m_rel = ifc_relationship
                break
        if ifc_m_rel is None:
            # Otherwise, if no existing relationship has been found create a new one.
            if ifc_rel.is_a('IfcRelDefinesByProperties'):
                ifc_m_rel = self._ifc_mm.IfcRelDefinesByProperties()
                # Add 'IfcPropertySetDefinitions' (handles 'IfcPropertySet' and 'IfcElementQuantity').
                ifc_m_property_set = self._add_ifc_property_set(ifc_model, ifc_rel.RelatingPropertyDefinition)
                if ifc_m_property_set is not None:
                    ifc_m_rel.relatingPropertyDefinition = ifc_m_property_set
            elif ifc_rel.is_a('IfcRelDefinesByType'):
                ifc_m_rel = self._ifc_mm.IfcRelDefinesByType()
                # Add 'IfcTypeObjects'.
                ifc_m_type_object = self._add_ifc_type_object(ifc_model, ifc_rel.RelatingType)
                ifc_m_rel.relatingType = ifc_m_type_object
        ifc_m_rel.relatedObjects.append(ifc_m_object)
        self._add_ifc_root_attributes(ifc_m_rel, ifc_rel)
        ifc_model.ifcRelationships.append(ifc_m_rel)

    def _add_ifc_property_set(self, ifc_m_element, ifc_property_set):
        """Adds 'IfcPropertySetDefinitions' to the given element. The term 'property set(s)' will be used synonymously
        for both, 'IfcPropertySet' and 'IfcElementQuantity'.

                :param ifc_m_element: The IFC modeling element for adding the property sets.
                :param ifc_property_set: The IFC element which holds the properties.

                :return: Returns a reference to the created modeling property set or 'None' for subtypes like
                'IfcDoorLiningProperties' to keep this prototype as simple as possible.
        """
```

```
        ifc_m_property_set = None
        if ifc_property_set.is_a('IfcPropertySet'):
            ifc_m_property_set = self._ifc_mm.IfcPropertySet()
            for ifc_property in ifc_property_set.HasProperties:
                if ifc_property.is_a('IfcSimpleProperty'):
                    self._add_ifc_simple_property(ifc_m_property_set, ifc_property)
                elif ifc_property.is_a('IfcComplexProperty'):
                    self._add_ifc_complex_property(ifc_m_property_set, ifc_property)
        elif ifc_property_set.is_a('IfcElementQuantity'):
            ifc_m_property_set = self._ifc_mm.IfcElementQuantity()
            if ifc_property_set.MethodOfMeasurement:
                ifc_m_property_set.methodOfMeasurement = ifc_property_set.MethodOfMeasurement
            for ifc_physical_quantity in ifc_property_set.Quantities:
                self._add_ifc_physical_quantity(ifc_m_property_set, ifc_physical_quantity)
        else:
            # Note: Some other 'IfcPropertySetDefinitions' like 'IfcDoorLiningProperties' will be ignored in this
            #  prototype, but should be included in a real implementation
            return ifc_m_property_set
        self._add_ifc_root_attributes(ifc_m_property_set, ifc_property_set)
        ifc_m_element.ifcPropertySets.append(ifc_m_property_set)
        return ifc_m_property_set

    def _add_ifc_simple_property(self, ifc_m_property_set, ifc_property):
        """Adds an 'IfcSimpleProperty' to the given property set.

                :param ifc_m_property_set: The IFC modeling property set for adding the property.
                :param ifc_property: The IFC element which holds the property.
        """
        ifc_m_simple_property = None
        # Each type of 'IfcSimpleProperty' has special attributes. (Note: Some of those attributes have been simplified
        #  to keep this prototype as simple as possible.)
        if ifc_property.is_a('IfcPropertySingleValue'):
            ifc_m_simple_property = self._ifc_mm.IfcPropertySingleValue()
            if ifc_property.NominalValue:
                ifc_m_simple_property.nominalValue = str(ifc_property.NominalValue.wrappedValue)
            if ifc_property.Unit:
                ifc_m_simple_property.unit = str(ifc_property.Unit.wrappedValue)
        elif ifc_property.is_a('IfcPropertyEnumeratedValue'):
            ifc_m_simple_property = self._ifc_mm.IfcPropertyEnumeratedValue()
            for enumerationValue in ifc_property.EnumerationValues:
                ifc_m_simple_property.enumerationValues.append(str(enumerationValue.wrappedValue))
            if ifc_property.EnumerationReference:
                ifc_m_simple_property.enumerationReference = ifc_property.EnumerationReference.Name
        elif ifc_property.is_a('IfcPropertyBoundedValue'):
            ifc_m_simple_property = self._ifc_mm.IfcPropertyBoundedValue()
            if ifc_property.UpperBoundValue:
                ifc_m_simple_property.upperBoundValue = str(ifc_property.UpperBoundValue.wrappedValue)
            if ifc_property.LowerBoundValue:
                ifc_m_simple_property.lowerBoundValue = str(ifc_property.LowerBoundValue.wrappedValue)
            if ifc_property.Unit:
```

```python
                ifc_m_simple_property.unit = str(ifc_property.Unit.wrappedValue)
        elif ifc_property.is_a('IfcPropertyTableValue'):
            ifc_m_simple_property = self._ifc_mm.IfcPropertyTableValue()
            for definingValue in ifc_property.DefiningValues:
                ifc_m_simple_property.definingValues.append(str(definingValue.wrappedValue))
            for definedValue in ifc_property.DefinedValues:
                ifc_m_simple_property.definedValues.append(str(definedValue.wrappedValue))
            if ifc_property.Expression:
                ifc_m_simple_property.expression = ifc_property.Expression
            if ifc_property.DefiningUnit:
                ifc_m_simple_property.definingUnit = str(ifc_property.DefiningUnit)
            if ifc_property.DefinedUnit:
                ifc_m_simple_property.definedUnit = str(ifc_property.DefinedUnit)
        elif ifc_property.is_a('IfcPropertyReferenceValue'):
            ifc_m_simple_property = self._ifc_mm.IfcPropertyReferenceValue()
            if ifc_property.UsageName:
                ifc_m_simple_property.usageName = ifc_property.UsageName
            ifc_m_simple_property.propertyReference = str(ifc_property.PropertyReference.wrappedValue)
        elif ifc_property.is_a('IfcPropertyListValue'):
            ifc_m_simple_property = self._ifc_mm.IfcPropertyListValue()
            for listValue in ifc_property.ListValues:
                ifc_m_simple_property.listValues.append(str(listValue.wrappedValue))
            if ifc_property.Unit:
                ifc_m_simple_property.unit = str(ifc_property.Unit.wrappedValue)
        ifc_m_simple_property.name = ifc_property.Name
        if ifc_property.Description:
            ifc_m_simple_property.description = ifc_property.Description
        ifc_m_property_set.hasProperties.append(ifc_m_simple_property)

    def _add_ifc_complex_property(self, ifc_m_property_set, ifc_property):
        """Adds an 'IfcComplexProperty' to the given property set.

                :param ifc_m_property_set: The IFC modeling property set for adding the property.
                :param ifc_property: The IFC element which holds the property.
        """
        ifc_m_complex_property = self._ifc_mm.IfcComplexProperty()
        ifc_m_complex_property.name = ifc_property.Name
        if ifc_property.Description:
            ifc_m_complex_property.description = ifc_property.Description
        ifc_m_complex_property.usageName = ifc_property.UsageName
        for ifc_sub_property in ifc_property.HasProperties:
            if ifc_sub_property.is_a('IfcSimpleProperty'):
                self._add_ifc_simple_property(ifc_m_complex_property, ifc_sub_property)
            elif ifc_sub_property.is_a('IfcComplexProperty'):
                self._add_ifc_complex_property(ifc_m_complex_property, ifc_sub_property)
        ifc_m_property_set.hasProperties.append(ifc_m_complex_property)

    def _add_ifc_physical_quantity(self, ifc_m_element_quantity, ifc_physical_quantity):
        """Adds an 'IfcPhysicalQuantity' to the given element quantity (property set for quantities).

                :param ifc_m_element_quantity: The IFC modeling element quantity for adding the quantity.
                :param ifc_physical_quantity: The IFC element which holds the quantity.
        """
        # Handling of 'IfcPhysicalQuantity' elements has been simplified to keep this
```

```
prototype as simple as possible.
        # Note: Exact handling should be analogous to processing of 'IfcProperty' elements and its subtypes.
        ifc_m_physical_quantity = self._ifc_mm.IfcPhysicalQuantity()
        ifc_m_physical_quantity.name = ifc_physical_quantity.Name
        if ifc_physical_quantity.Description:
            ifc_m_physical_quantity.description = ifc_physical_quantity.Description
        ifc_m_element_quantity.quantities.append(ifc_m_physical_quantity)

    def _add_ifc_type_object(self, ifc_model, ifc_type_object):
        """Adds an 'IfcTypeObject' to the given model element and with respect to its relationship.

                :param ifc_model: The IFC model (root) element.
                :param ifc_type_object: The IFC element which holds the type object information.

                :return: Returns a reference to the created modeling element for further usage.
        """
        ifc_m_type_object = self._ifc_mm.IfcTypeObject()
        self._add_ifc_root_attributes(ifc_m_type_object, ifc_type_object)
        if ifc_type_object.ApplicableOccurrence:
            ifc_m_type_object.applicableOccurrence = ifc_type_object.ApplicableOccurrence
        if ifc_type_object.HasPropertySets:
            for ifc_property_set in ifc_type_object.HasPropertySets:
                self._add_ifc_property_set(ifc_m_type_object, ifc_property_set)
        ifc_model.ifcTypeObjects.append(ifc_m_type_object)
        return ifc_m_type_object

    def _save_ifc_model(self, ifc_model):
        """Saves the IFC model as a '.ifcmodel' file.

                :param ifc_model: The IFC model (root) element.
        """
        ifc_m_rset = ResourceSet()
        ifc_m_filename = Path(self._ifc_filename).stem
        if self._view_name is not None:
        ifc_m_filename += '_' + self._view_name.lower()
        ifc_m_resource = ifc_m_rset.create_resource(URI('../ifcr.examples/{}.ifcmodel'.format(ifc_m_filename)))
        ifc_m_resource.append(ifc_model)
        ifc_m_resource.save()


def calculate_creation_time(start_time, end_time):
    """Calculates and prints out the duration of the whole creation process.

            :param start_time: Start time of the process.
            :param end_time: End time of the process.
    """
    t_sec = round(end_time - start_time)
    (t_min, t_sec) = divmod(t_sec, 60)
    print("Creation of IFC model finished in: {} minute(s) {} second(s).".format(t_min, t_sec))


def main():
    ifc_model_creator = IfcModelCreator()
```

```
    print("Creating the IFC model...")
    try:
        start_time = process_time()
        ifc_model_creator.create_ifc_model()
        end_time = process_time()
        calculate_creation_time(start_time, end_time)
    except Exception as e:
        sys.exit("An error occurred while creating the IFC model! Error: \n\t%s" % e)


if __name__ == '__main__':
    main()
```

Listing B.2: IFC2CROM.etl

```
pre {
    // TODO: Set correct encoding -> UTF-8 instead of ASCII (if possible).
    "Starting IFC2CROM transformation...".printInfo();
    // Define 'cromModel' globally for availability in all rules. Also add default types to the model.
    var cromModel : new CROM!Model;
    cromModel.addDefaultDataTypes();
    cromModel.addDefaultNaturalTypes();
    cromModel.addDefaultCompartmentTypes();
    // (Verbose) Logging
    var totalObjectCount = IFCModel!IfcObject.all.size;
    totalObjectCount.format("%1$d IfcObject(s) will be transformed in total.").printInfo();
    var objectCount = 0;
    var startTime = Native('java.lang.System').currentTimeMillis;
}

post {
    var endTime = Native('java.lang.System').currentTimeMillis;
    (endTime - startTime).format("Finished IFC2CROM transformation in: %1$tM minute(s) %1$tS second(s).").printInfo();
}

rule IfcObjectModel2IfcObjectNT
transform ifcObjectModel : IFCModel!IfcObject
to ifcObjectNT : CROM!NaturalType {
    if (not CROM!NaturalType.all.exists(nt|nt.name = ifcObjectModel.subtype)) {
        // TODO: Name should be derived from 'ifcObjectModel.objectType' if set, otherwise 'ifcObjectModel.subtype' will be taken.
        //  This is especially important for the semantic of objects of type 'IfcBuildingElementProxy'.
        ifcObjectNT.name = ifcObjectModel.subtype;
        cromModel.elements.add(ifcObjectNT);
        cromModel.relations.add(new CROM!NaturalInheritance(super = retrieveDefaultType(CROM!NaturalType, 'IfcObject'), sub = ifcObjectNT));
    } else {
        delete ifcObjectNT;
    }
    objectCount++;
    if (objectCount.mod(1000) = 0) {
        objectCount.format("%1$d IfcObject(s) out of ").concat(totalObjectCount.format("%1$d processed.")).printInfo();
    }
}

```

```
rule IfcTypeObjectModel2IfcTypeObjectRT
transform ifcTypeObjectModel : IFCModel!IfcTypeObject
to ifcTypeObjectRT : CROM!RoleType {
    var ifcTypeObjectName = retrieveSpecificationName(ifcTypeObjectModel);
    if (not CROM!RoleType.all.exists(rt|rt.name = ifcTypeObjectName)) {
        ifcTypeObjectRT.name = ifcTypeObjectName;
        var ifcTypeObjectPart : new CROM!Part(lower = 0, upper = -1);
        ifcTypeObjectPart.role = ifcTypeObjectRT;
        var objectDefinition = retrieveDefaultType(CROM!CompartmentType, 'ObjectDefinition');
        objectDefinition.parts.add(ifcTypeObjectPart);
        cromModel.relations.add(new CROM!RoleInheritance(super = retrieveDefaultType(CROM!RoleType, 'IfcTypeObject'), sub = ifcTypeObjectRT));
    } else {
        delete ifcTypeObjectRT;
    }
}

rule IfcPropertySetModel2IfcPropertySetRT
transform ifcPropertySetModel : IFCModel!IfcPropertySet
to ifcPropertySetRT : CROM!RoleType {
    guard : ifcPropertySetModel.eContainer.eClass.name <> 'IfcTypeObject'
    var ifcPropertySetName = retrieveSpecificationName(ifcPropertySetModel);
    if (not CROM!RoleType.all.exists(rt|rt.name = ifcPropertySetName)) {
        ifcPropertySetRT.name = ifcPropertySetName;
        var ifcPropertySetPart : new CROM!Part(lower = 0, upper = -1);
        ifcPropertySetPart.role = ifcPropertySetRT;
        var objectDefinition = retrieveDefaultType(CROM!CompartmentType, 'ObjectDefinition');
        objectDefinition.parts.add(ifcPropertySetPart);
        cromModel.relations.add(new CROM!RoleInheritance(super = retrieveDefaultType(CROM!RoleType, 'IfcPropertySet'), sub = ifcPropertySetRT));
    } else {
        delete ifcPropertySetRT;
    }
}

rule IfcElementQuantityModel2IfcElementQuantityRT
transform ifcElementQuantityModel : IFCModel!IfcElementQuantity
to ifcElementQuantityRT : CROM!RoleType {
    guard : ifcElementQuantityModel.eContainer.eClass.name <> 'IfcTypeObject'
    var ifcElementQuantityName = retrieveSpecificationName(ifcElementQuantityModel);
    if (not CROM!RoleType.all.exists(rt|rt.name = ifcElementQuantityName)) {
        ifcElementQuantityRT.name = ifcElementQuantityName;
        var ifcElementQuantityPart : new CROM!Part(lower = 0, upper = -1);
        ifcElementQuantityPart.role = ifcElementQuantityRT;
        var objectDefinition = retrieveDefaultType(CROM!CompartmentType, 'ObjectDefinition');
        objectDefinition.parts.add(ifcElementQuantityPart);
        cromModel.relations.add(new CROM!RoleInheritance(super = retrieveDefaultType(CROM!RoleType, 'IfcElementQuantity'), sub = ifcElementQuantityRT));
    } else {
        delete ifcElementQuantityRT;
    }
}

rule ViewInformation2ViewInformationDT
transform viewInformation : IFCModel!ViewInformation
to viewInformationDT : CROM!DataType {
```

```
    viewInformationDT.name = 'VI_' + viewInformation.key;
    cromModel.elements.add(viewInformationDT);
}

operation Any addIfcRootAttributes() {
    self.attributes.add(new CROM!Attribute(name = 'globalId', type = retrieveDefaultType(CROM!DataType, 'IfcGloballyUniqueId')));
    self.attributes.add(new CROM!Attribute(name = 'name'));
    self.attributes.add(new CROM!Attribute(name = 'description'));
}

operation CROM!Model addDefaultDataTypes() {
    // Data Type: IfcGloballyUniqueId
    var ifcGloballyUniqueId : new CROM!DataType(name = 'IfcGloballyUniqueId');
    self.elements.add(ifcGloballyUniqueId);
    // Data Type: IfcProperty
    var ifcProperty : new CROM!DataType(name = 'IfcProperty');
    ifcProperty.attributes.add(new CROM!Attribute(name = 'name'));
    ifcProperty.attributes.add(new CROM!Attribute(name = 'description'));
    self.elements.add(ifcProperty);
    // Data Type: IfcComplexProperty
    var ifcComplexProperty : new CROM!DataType(name = 'IfcComplexProperty');
    ifcComplexProperty.attributes.add(new CROM!Attribute(name = 'usageName'));
    ifcComplexProperty.attributes.add(new CROM!Attribute(name = 'hasProperties', type = ifcProperty));
    self.elements.add(ifcComplexProperty);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcComplexProperty));
    // Data Type: IfcPropertySingleValue
    var ifcPropertySingleValue : new CROM!DataType(name = 'IfcPropertySingleValue');
    ifcPropertySingleValue.attributes.add(new CROM!Attribute(name = 'nominalValue'));
    ifcPropertySingleValue.attributes.add(new CROM!Attribute(name = 'unit'));
    self.elements.add(ifcPropertySingleValue);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcPropertySingleValue));
    // Data Type: IfcPropertyEnumeratedValue
    var ifcPropertyEnumeratedValue : new CROM!DataType(name = 'IfcPropertyEnumeratedValue');
    ifcPropertyEnumeratedValue.attributes.add(new CROM!Attribute(name = 'enumerationValues'));
    ifcPropertyEnumeratedValue.attributes.add(new CROM!Attribute(name = 'enumerationReference'));
    self.elements.add(ifcPropertyEnumeratedValue);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcPropertyEnumeratedValue));
    // Data Type: IfcPropertyBoundedValue
    var ifcPropertyBoundedValue : new CROM!DataType(name = 'IfcPropertyBoundedValue');
    ifcPropertyBoundedValue.attributes.add(new CROM!Attribute(name = 'upperBoundValue'));
    ifcPropertyBoundedValue.attributes.add(new CROM!Attribute(name = 'lowerBoundValue'));
    ifcPropertyBoundedValue.attributes.add(new CROM!Attribute(name = 'unit'));
    self.elements.add(ifcPropertyBoundedValue);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcPropertyBoundedValue));
    // Data Type: IfcPropertyTableValue
    var ifcPropertyTableValue : new CROM!DataType(name = 'IfcPropertyTableValue');
    ifcPropertyTableValue.attributes.add(new CROM!Attribute(name = 'definingValues'));
    ifcPropertyTableValue.attributes.add(new CROM!Attribute(name = 'definedValues'));
    ifcPropertyTableValue.attributes.add(new CROM!Attribute(name = 'expression'));
    ifcPropertyTableValue.attributes.add(new CROM!Attribute(name = 'definingUnit'));
```

```
    ifcPropertyTableValue.attributes.add(new CROM!Attribute(name = 'definedUnit'));
    self.elements.add(ifcPropertyTableValue);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcPropertyTableValue));
    // Data Type: IfcPropertyReferenceValue
    var ifcPropertyReferenceValue : new CROM!DataType(name = 'IfcPropertyReferenceValue');
    ifcPropertyReferenceValue.attributes.add(new CROM!Attribute(name = 'usageName'));
    ifcPropertyReferenceValue.attributes.add(new CROM!Attribute(name = 'propertyReference'));
    self.elements.add(ifcPropertyReferenceValue);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcPropertyReferenceValue));
    // Data Type: IfcPropertyListValue
    var ifcPropertyListValue : new CROM!DataType(name = 'IfcPropertyListValue');
    ifcPropertyListValue.attributes.add(new CROM!Attribute(name = 'listValues'));
    ifcPropertyListValue.attributes.add(new CROM!Attribute(name = 'unit'));
    self.elements.add(ifcPropertyListValue);
    self.relations.add(new CROM!DataInheritance(super = ifcProperty, sub = ifcPropertyListValue));
    // Data Type: IfcPhysicalQuantity
    var ifcPhysicalQuantity : new CROM!DataType(name = 'IfcPhysicalQuantity');
    ifcPhysicalQuantity.attributes.add(new CROM!Attribute(name = 'name'));
    ifcPhysicalQuantity.attributes.add(new CROM!Attribute(name = 'description'));
    self.elements.add(ifcPhysicalQuantity);
}

operation CROM!Model addDefaultNaturalTypes() {
    // Natural Type: IfcObject
    var ifcObject : new CROM!NaturalType(name = 'IfcObject');
    ifcObject.addIfcRootAttributes();
    // TODO: Property 'objectType' only needed for naming purposes if set, e.g. an 'IfcDoor' will be a 'FireDoor' if set as 'objectType'.
    // ifcObject.attributes.add(new CROM!Attribute(name = 'objectType'));
    self.elements.add(ifcObject);
}

operation CROM!Model addDefaultCompartmentTypes() {
    // Compartment Type: ObjectDefinition
    var objectDefinition : new CROM!CompartmentType(name = 'ObjectDefinition');
    objectDefinition.addDefaultRoles();
    self.elements.add(objectDefinition);
}

operation CROM!CompartmentType addDefaultRoles() {
    // Role Type: IfcTypeObject
    var ifcTypeObjectPart : new CROM!Part(lower = 0, upper = -1);
    var ifcTypeObjectRole : new CROM!RoleType(name = 'IfcTypeObject');
    ifcTypeObjectRole.addIfcRootAttributes();
    ifcTypeObjectRole.attributes.add(new CROM!Attribute(name = 'applicableOccurrence'));
    ifcTypeObjectPart.role = ifcTypeObjectRole;
    self.parts.add(ifcTypeObjectPart);
    cromModel.relations.add(new CROM!Fulfillment(filled = ifcTypeObjectRole, filler = retrieveDefaultType(CROM!NaturalType, 'IfcObject')));
    // Role Type: IfcPropertySetDefinition
    var ifcPropertySetDefinitionPart : new CROM!Part(lower = 0, upper = -1);
    var ifcPropertySetDefinitionRole : new CROM!RoleType(name = 'IfcPropertySetDefinition');
    ifcPropertySetDefinitionRole.addIfcRootAttributes();
    ifcPropertySetDefinitionPart.role = ifcPropertySetDefinitionRole;
```

```
    self.parts.add(ifcPropertySetDefinitionPart);
    cromModel.relations.add(new CROM!Fulfillment(filled = ifcPropertySetDefinitionRole, filler = retrieveDefaultType(CROM!NaturalType, 'IfcObject')));
    // Role Type: IfcPropertySet
    var ifcPropertySetPart : new CROM!Part(lower = 0, upper = -1);
    var ifcPropertySetRole : new CROM!RoleType(name = 'IfcPropertySet');
    ifcPropertySetRole.attributes.add(new CROM!Attribute(name = 'hasProperties', type = retrieveDefaultType(CROM!DataType, 'IfcProperty')));
    ifcPropertySetPart.role = ifcPropertySetRole;
    self.parts.add(ifcPropertySetPart);
    cromModel.relations.add(new CROM!RoleInheritance(super = ifcPropertySetDefinitionRole, sub = ifcPropertySetRole));
    // Role Type: IfcElementQuantity
    var ifcElementQuantityPart : new CROM!Part(lower = 0, upper = -1);
    var ifcElementQuantityRole : new CROM!RoleType(name = 'IfcElementQuantity');
    ifcElementQuantityRole.attributes.add(new CROM!Attribute(name = 'methodOfMeasurement'));
    ifcElementQuantityRole.attributes.add(new CROM!Attribute(name = 'quantities', type = retrieveDefaultType(CROM!DataType, 'IfcPhysicalQuantity')));
    ifcElementQuantityPart.role = ifcElementQuantityRole;
    self.parts.add(ifcElementQuantityPart);
    cromModel.relations.add(new CROM!RoleInheritance(super = ifcPropertySetDefinitionRole, sub = ifcElementQuantityRole));
}

operation retrieveDefaultType(cromType : Any, typeName : String) {
    return cromType.all.selectOne(t|t.name = typeName);
}

operation retrieveSpecificationName(ifcModelElement : Any) {
    var specificationName = 'None';
    if (ifcModelElement.name.isDefined()) {
        specificationName = ifcModelElement.name.replace('[^a-zA-Z]', '');
    }
    if (ifcModelElement.name.isUndefined() or specificationName.length <= 0) {
        if (ifcModelElement.subtype.isDefined()) {
            specificationName = ifcModelElement.subtype;
        } else {
            specificationName = ifcModelElement.eClass.name;
        }
    }
    // Prefix needed because the name of an 'IfcPropertySet' could be taken for an 'IfcElementQuantity' as well.
    var prefix;
    switch (ifcModelElement.eClass.name) {
        case 'IfcTypeObject' : prefix = 'T_';
        case 'IfcPropertySet' : prefix = 'PS_';
        case 'IfcElementQuantity' : prefix = 'EQ_';
        default : prefix = '';
    }
    return prefix.concat(specificationName);
}

operation Any printInfo() {
    return self.println('[INFO] ');
}

operation Integer mod(i : Integer) {
    return self - (self/i * i);
```

```
}
```

Listing B.3: IFC2CROI.etl

```
pre {
    "Starting IFC2CROI transformation...".printInfo();
    // (Verbose) Logging
    var totalObjectCount = IFCModel!IfcObject.all.size;
    totalObjectCount.format("%1$d IfcObject(s) will be transformed in total.").printInfo();
    var objectCount = 0;
    var objectDefinitionCount = 1;
    var startTime = Native('java.lang.System').currentTimeMillis;
}

post {
    var endTime = Native('java.lang.System').currentTimeMillis;
    (endTime - startTime).format("Finished IFC2CROI transformation in: %1$tM minute(s) %1
    $tS second(s).").printInfo();
}

rule IFC2CROI
transform ifcModel : IFCModel!Model
to croi : CROI!t_croi {
    CROI.root = croi;
    // IfcObjects
    var ifcObjects : new CROI!t_ifcObjects;
    croi.appendChild(ifcObjects);
    var ifcObjectModels = ifcModel.ifcObjects;
    if (ifcObjectModels.isDefined() and ifcObjectModels.notEmpty()) {
        // ObjectDefinitions are only needed if at least one 'IfcObject' is defined further
     by properties or types.
        if (ifcObjectModels.exists(ifcObject|ifcObject.isDefinedBy.isDefined())) {
            // ObjectDefinitions
            var objectDefinitions : new CROI!t_objectDefinitions;
            croi.appendChild(objectDefinitions);
            // Iterate over all 'IfcObjects' and handle their properties and relationships.
            ifcObjects.handleIfcObjects(ifcObjectModels);
        }
    }
    var viewInformationModels = ifcModel.viewInformation;
    if (viewInformationModels.isDefined() and viewInformationModels.notEmpty()) {
        // Add view information to the CROI if available.
        var viewInformation : new CROI!t_viewInformation;
        viewInformation.handleViewInformation(viewInformationModels);
        croi.appendChild(viewInformation);
    }
}

operation CROI!t_ifcObjects handleIfcObjects(ifcObjectModels : Collection) {
    for (ifcObjectModel in ifcObjectModels) {
        // Create natural but only if the corresponding natural type exists in the CROM.
        if (CROM!NaturalType.all.exists(nt|nt.name = ifcObjectModel.subtype)) {
            var ifcObjectN = CROI.createInstance('t_' + ifcObjectModel.subtype);
            ifcObjectN.addAttributesForType(ifcObjectModel, CROM!NaturalType, 'IfcObject');
            if (ifcObjectN.a_name.isUndefined()) {
                ifcObjectN.a_name = retrieveElementName(ifcObjectModel);
            }
            self.appendChild(ifcObjectN);
            // IfcRelationships (handled as 'ObjectDefinitions')
```

```
            CROI!t_objectDefinitions.all.first.handleIfcRelationships(ifcObjectModel);
            objectCount++;
            if (objectCount.mod(1000) = 0) {
                objectCount.format("%1$d IfcObject(s) out of ").concat(totalObjectCount.format("%1$d processed.")).printInfo();
            }
        }
    }
}

operation CROI!t_objectDefinitions handleIfcRelationships(ifcObjectModel : IFCModel!IfcObject) {
    var ifcRelationships = ifcObjectModel.isDefinedBy;
    if (ifcRelationships.isDefined() and ifcRelationships.notEmpty()) {
        var objectDefinition : new CROI!t_objectDefinition(a_name = 'od' + objectDefinitionCount);
        objectDefinition.appendChild(new CROI!t_player(a_referenceId = ifcObjectModel.globalId));
        objectDefinition.appendChild(new CROI!t_plays);
        for (ifcRelationship in ifcRelationships) {
            switch (ifcRelationship.eClass.name) {
                case 'IfcRelDefinesByProperties' : {
                    objectDefinition.handleIfcPropertySets(ifcRelationship.relatingPropertyDefinition);
                }
                case 'IfcRelDefinesByType' : {
                    objectDefinition.handleIfcTypeObjects(ifcRelationship.relatingType);
                }
            }
        }
        // Attribute 'origin' was only needed to identify the origin of a 'IfcPropertySet' and, therefore, is not needed in the actual CROI.
        for (role in objectDefinition.e_plays.children) {
            if (role.a_origin.isDefined()) {
                role.removeAttribute('origin');
            }
        }
        self.appendChild(objectDefinition);
        objectDefinitionCount++;
    }
}

operation CROI!t_objectDefinition handleIfcPropertySets(relatingPropertyDefinition : IFCModel!IfcPropertySetDefinition) {
    var ifcPropertySetSpecName = retrieveSpecificationName(relatingPropertyDefinition);
    // Create role but only if the corresponding role type exists in the CROM or the 'IfcPropertySet' is part of an 'IfcTypeObject'.
    if (CROM!RoleType.all.exists(rt|rt.name = ifcPropertySetSpecName or relatingPropertyDefinition.eContainer.eClass.name = 'IfcTypeObject')) {
        var ifcPropertySetR = CROI.createInstance('t_' + ifcPropertySetSpecName);
        ifcPropertySetR.addAttributesForType(relatingPropertyDefinition, CROM!RoleType, 'IfcPropertySetDefinition');
        if (ifcPropertySetR.a_name.isUndefined()) {
            ifcPropertySetR.a_name = retrieveElementName(relatingPropertyDefinition);
        }
        // Origin of the 'IfcPropertySet' is needed for merging property sets of occurrences with property sets of 'IfcTypeObjects'.
        ifcPropertySetR.a_origin = relatingPropertyDefinition.eContainer.eClass.name;
        switch (relatingPropertyDefinition.eClass.name) {
```

```
            case 'IfcPropertySet' : {
                ifcPropertySetR.addAttributesForType(relatingPropertyDefinition, CROM!
    RoleType, 'IfcPropertySet');
            }
            case 'IfcElementQuantity' : {
                ifcPropertySetR.addAttributesForType(relatingPropertyDefinition, CROM!
    RoleType, 'IfcElementQuantity');
            }
        }
        self.mergeIfcPropertySets(ifcPropertySetR);
    }
}

operation CROI!t_objectDefinition handleIfcTypeObjects(relatingType : IFCModel!
    IfcTypeObject) {
    var ifcTypeObjectSpecName = retrieveSpecificationName(relatingType);
    // Create role but only if the corresponding role type exists in the CROM.
    if (CROM!RoleType.all.exists(rt|rt.name = ifcTypeObjectSpecName)) {
        var ifcTypeObjectR = CROI.createInstance('t_' + ifcTypeObjectSpecName);
        ifcTypeObjectR.addAttributesForType(relatingType, CROM!RoleType, 'IfcTypeObject');
        if (ifcTypeObjectR.a_name.isUndefined()) {
            ifcTypeObjectR.a_name = retrieveElementName(relatingType);
        }
        self.e_plays.appendChild(ifcTypeObjectR);
        var hasPropertySets = relatingType.hasPropertySets;
        if (hasPropertySets.isDefined() and hasPropertySets.notEmpty()) {
            for (propertySet in hasPropertySets) {
                self.handleIfcPropertySets(propertySet);
            }
        }
    }
}

operation CROI!t_viewInformation handleViewInformation(viewInformationModels : Collection)
    {
    for (viewInformation in viewInformationModels) {
        // Add view information but only if the corresponding data type exists in the CROM.
        var viewInformationSpecName = retrieveSpecificationName(viewInformation);
        if (CROM!DataType.all.exists(dt|dt.name = viewInformationSpecName)) {
            var viewInformationD = CROI.createInstance('t_' + viewInformationSpecName);
            viewInformationD.a_value = viewInformation.value;
            self.appendChild(viewInformationD);
        }
    }
}

operation CROI!t_objectDefinition mergeIfcPropertySets(currentPropertySet : Any) {
    // 'IfcPropertySets' of 'IfcTypeObjects' need to be merged with already existing
    property sets with the same name according to the IFC.
    // Note: For further information check 'https://standards.buildingsmart.org/IFC/DEV/
    IFC4_2/FINAL/HTML/link/ifcreldefinesbytype.htm'.
    if (self.e_plays.children.isEmpty()) {
        self.e_plays.appendChild(currentPropertySet);
    } else {
        var existingPropertySets = self.e_plays.children.select(child|not child.name.
    startsWith('T_'));
        if (not existingPropertySets.exists(child|child.name = currentPropertySet.name)) {
            self.e_plays.appendChild(currentPropertySet);
        } else {
```

```
            // TODO: 'globalId' of an 'IfcPropertySet' of an 'IfcTypeObject' must be changed to the overwriting
            //  'IfcPropertySet's 'globalId' or by creating a new one. This will be needed for back transformation.
            var existingPropertySet = existingPropertySets.selectOne(pset|pset.name = currentPropertySet.name);
            for (currentProperty in currentPropertySet.children) {
                if (not existingPropertySet.children.exists(existingProperty|existingProperty.a_name = currentProperty.a_name)) {
                    existingPropertySet.appendChild(currentProperty);
                } else {
                    if (existingPropertySet.a_origin = 'IfcTypeObject') {
                        var existingProperty = existingPropertySet.children.selectOne(existingProperty|existingProperty.a_name = currentProperty.a_name);
                        if (existingProperty.name == 'IfcComplexProperty') {
                            for (currentChildProperty in currentProperty.children) {
                                if (not existingProperty.children.exists(childProperty|childProperty.a_name == currentChildProperty.a_name)) {
                                    existingProperty.appendChild(currentChildProperty);
                                } else {
                                    var existingChildProperty = existingProperty.children.selectOne(childProperty|childProperty.a_name == currentChildProperty.a_name);
                                    existingChildProperty.mergeIfcProperties(currentChildProperty);
                                }
                            }
                        } else {
                            existingProperty.mergeIfcProperties(currentProperty);
                        }
                    }
                }
            }
        }
    }
}

operation Any mergeIfcProperties(currentProperty : Any) {
    var currentPropertyAttributes = currentProperty.attributes;
    var i = 0;
    while (i < currentPropertyAttributes.length) {
        var currentPropertyAttribute = currentPropertyAttributes.item(i);
        if (self.getAttribute(currentPropertyAttribute.name) <> currentPropertyAttribute.value) {
            self.setAttribute(currentPropertyAttribute.name, currentPropertyAttribute.value);
        }
        i++;
    }
}

operation Any addAttributesForType(ifcModelElement : Any, cromType : Any, typeName : String) {
    // Add attributes and values to the natural according to the given CROM.
    var ifcMetaElement = cromType.all.selectOne(t|t.name = typeName);
    for (attribute in ifcMetaElement.attributes) {
        var attributeValue = ifcModelElement.eGet(ifcModelElement.eClass.getEStructuralFeature(attribute.name));
        if ((attributeValue.isKindOf(Collection) and attributeValue.notEmpty()) or attributeValue.isDefined()) {
```

```
            switch (attribute.name) {
                case 'hasProperties' : {
                    for (ifcProperty in attributeValue) {
                        self.addProperties(ifcProperty);
                    }
                }
                case 'quantities' : {
                    for (quantity in attributeValue) {
                        self.addQuantities(quantity);
                    }
                }
                default : {
                    self.setAttribute(attribute.name, attributeValue);
                }
            }
        }
    }
}

operation Any addProperties(ifcProperty : IFCModel!IfcProperty) {
    var ifcPropertySpecName = retrieveSpecificationName(ifcProperty);
    var ifcPropertyD = CROI.createInstance('t_' + ifcPropertySpecName);
    ifcPropertyD.addAttributesForType(ifcProperty, CROM!DataType, 'IfcProperty');
    ifcPropertyD.addAttributesForType(ifcProperty, CROM!DataType, ifcProperty.eClass.name);
    self.appendChild(ifcPropertyD);
}

operation Any addQuantities(ifcPhysicalQuantity : IFCModel!IfcPhysicalQuantity) {
    // Handling of 'IfcPhysicalQuantity' elements has been simplified to keep this prototype as simple as possible.
    // Note: Exact handling should be analogous to processing of 'IfcProperty' elements and its subtypes.
    var ifcPhysicalQuantitySpecName = retrieveSpecificationName(ifcPhysicalQuantity);
    var ifcPhysicalQuantityD = CROI.createInstance('t_' + ifcPhysicalQuantitySpecName);
    ifcPhysicalQuantityD.addAttributesForType(ifcPhysicalQuantity, CROM!DataType, 'IfcPhysicalQuantity');
    self.appendChild(ifcPhysicalQuantityD);
}

operation retrieveElementName(ifcModelElement : Any) {
    var elementName = 'None';
    if (ifcModelElement.name.isDefined()) {
        elementName = ifcModelElement.name;
    } else if (ifcModelElement.subtype.isDefined()) {
        elementName = ifcModelElement.subtype;
    } else {
        elementName = ifcModelElement.eClass.name;
    }
    return elementName;
}

operation retrieveSpecificationName(ifcModelElement : Any) {
    var specificationName = 'None';
    if (ifcModelElement.eClass.EAllSuperTypes.exists(st|st.name = 'IfcProperty')) {
        specificationName = ifcModelElement.eClass.name.replace('[^a-zA-Z]', '');
    } else if (ifcModelElement.eClass.name = 'IfcPhysicalQuantity') {
        specificationName = ifcModelElement.eClass.name.replace('[^a-zA-Z]', '');
    } else if (ifcModelElement.eClass.name = 'ViewInformation') {
        specificationName = ifcModelElement.key;
```

```
    } else {
        specificationName = retrieveElementName(ifcModelElement).replace('[^a-zA-Z]', '');
        if (specificationName.length <= 0) {
            if (ifcModelElement.subtype.isDefined()) {
                specificationName = ifcModelElement.subtype;
            } else {
                specificationName = ifcModelElement.eClass.name;
            }
        }
    }
    // Prefix needed because the name of an 'IfcPropertySet' could be taken for an 'IfcElementQuantity' as well.
    var prefix;
    switch (ifcModelElement.eClass.name) {
        case 'IfcTypeObject' : prefix = 'T_';
        case 'IfcPropertySet' : prefix = 'PS_';
        case 'IfcElementQuantity' : prefix = 'EQ_';
        case 'ViewInformation' : prefix = 'VI_';
        default : prefix = '';
    }
    return prefix.concat(specificationName);
}

operation Any printInfo() {
    return self.println('[INFO] ');
}

operation Integer mod(i : Integer) {
    return self - (self/i * i);
}
```

Listing B.4: view_extensions.py

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

"""View extensions for the creation of IFC models (preprocessing step).

This script contains all extension methods for extending the domain view mechanism of the preprocessing step of creating
an IFC model for the IFC2CROM/IFC2CROI transformations. New extension methods should be added according to the existing
view extensions to maintain proper framework extensibility.

Note (1): Extensions will be included by calling the corresponding method via 'getattr()'. Each extension needs to be
configured according to the documentation ('README').

Note (2): 'remove_' methods (general functions which modify the input IFC/STEP file) should be applied first.

Note (3): Methods starting with 'filter_' will append the list of added 'IfcObjects'.

Note (4): Methods starting with 'vi_' will append the list of view information (Metamodel/EClass 'ViewInformation').
Therefore, it is required for such methods to return two values: The key and the value as Strings of the view
information.

```

```python
"""

import re

from itertools import filterfalse

__author__ = "Martin Klaude"
__status__ = "Prototype"


class ViewExtensions:
    def __init__(self, ifc_file):
        self._ifc_file = ifc_file
        self._all_ifc_objects = self._ifc_file.by_type('IfcObject')

    @staticmethod
    def method_not_found():
        """Method informing about wrong usage of view extensions. """
        print("The given extension method was not found.")

    def remove_ifc_property_sets(self, del_ifc_property_sets):
        """Removes the given 'IfcPropertySets' and 'IfcElementQuantities' from the IFC model.

                :param del_ifc_property_sets: A list of 'IfcPropertySet' and 'IfcElementQuantities' names which should not
                be added to the model.
        """
        ifc_property_sets = self._ifc_file.by_type('IfcPropertySet')
        ifc_property_sets.extend(self._ifc_file.by_type('IfcElementQuantity'))
        for ifc_property_set in ifc_property_sets:
            if ifc_property_set.Name and ifc_property_set.Name in del_ifc_property_sets:
                property_definition_of = ifc_property_set.PropertyDefinitionOf[0]
                # The order is important here because removing the 'IfcRelDefines' element first would modify the
                #  reference in memory of the 'RelatingPropertyDefinition' but not the element itself from the
                #  IFC/STEP file.
                self._ifc_file.remove(ifc_property_set)
                self._ifc_file.remove(property_definition_of)

    def vi_aggregate_numeric_ifc_property_single_value(self, key, ifc_property_name):
        """Simple example of a method aggregating the 'NominalValues' of a certain 'IfcPropertySingleValue' (by name).

                :param key: Key for storing the view information in the IFC model.
                :param ifc_property_name: Name of the 'IfcPropertySingleValue' which should be aggregated.

                :return: The given key for the view information dict and the aggregated value as String.
        """
        aggregated_value = 0
        for ifc_property in self._ifc_file.by_type('IfcPropertySingleValue'):
            if ifc_property.Name == ifc_property_name:
                value = ifc_property.NominalValue.wrappedValue
                if value.isdecimal():
                    # Handle the value as simple integer.
                    value = int(value)
```

```
                elif self._is_float(value):
                    # Handle the value as float.
                    value = float(value)
                aggregated_value += value
        return key, str(aggregated_value)

    def filter_ifc_objects(self, ifc_objects_filter):
        """Only 'IfcObjects' listed in the filter will be added to the IFC model.

                :param ifc_objects_filter: A list of subtypes of 'IfcObject' which should be added to the model.

                :return: A filtered list of 'IfcObjects'.
        """
        filtered_ifc_objects = self._all_ifc_objects.copy()
        filtered_ifc_objects[:] = filterfalse(
            lambda ifc_object: ifc_object.is_a() not in ifc_objects_filter, filtered_ifc_objects)
        return filtered_ifc_objects

    def filter_ifc_property_sets(self, ifc_property_sets_filter):
        """Only 'IfcObjects' containing at least one of the given 'IfcPropertySets' and 'IfcElementQuantities' (by name)
        will be added to the IFC model.

                :param ifc_property_sets_filter: A list of 'IfcPropertySet' and 'IfcElementQuantities' names to search for
                in the 'IfcObjects'.

                :return: A filtered list of 'IfcObjects'.
        """
        filtered_ifc_objects = self._all_ifc_objects.copy()
        filtered_ifc_objects[:] = filterfalse(
            lambda ifc_object: not self._has_property_sets(ifc_object, ifc_property_sets_filter), filtered_ifc_objects)
        return filtered_ifc_objects

    def filter_ifc_type_object(self, ifc_type_objects_filter):
        """Only 'IfcObjects' typed by at least one of the given 'IfcTypeObjects' (by name) will be added to the IFC
        model.

                :param ifc_type_objects_filter: A list of 'IfcTypeObject' names to search for in the 'IfcObjects'.

                :return: A filtered list of 'IfcObjects'.
        """
        filtered_ifc_objects = self._all_ifc_objects.copy()
        filtered_ifc_objects[:] = filterfalse(
            lambda ifc_object: not self._has_type_objects(ifc_object, ifc_type_objects_filter), filtered_ifc_objects)
        return filtered_ifc_objects

    def filter_for_fire_prevention(self):
        """Simple example of a filter creating a view for the stakeholder controlling the fire safety regulations.

                :return: A filtered list of 'IfcObjects'.
        """
```

```python
        filtered_ifc_objects = []
        fire_regex = r'.*fire.*'
        for ifc_object in self._all_ifc_objects.copy():
            add_object = False
            # Check 'IfcPropertySets', 'IfcProperties' and 'IfcTypeObjects' for names containing '*fire*'.
            for ifc_rel in ifc_object.IsDefinedBy:
                if ifc_rel.is_a('IfcRelDefinesByProperties'):
                    ifc_property_set = ifc_rel.RelatingPropertyDefinition
                    if re.match(fire_regex, ifc_property_set.Name, re.IGNORECASE):
                        add_object = True
                        break
                if ifc_property_set.is_a('IfcPropertySet'):
                    for ifc_property in ifc_property_set.HasProperties:
                        if re.match(fire_regex, ifc_property.Name, re.IGNORECASE):
                            add_object = True
                            break
                elif ifc_property_set.is_a('IfcElementQuantity'):
                    for ifc_physical_quantity in ifc_property_set.Quantities:
                        if re.match(fire_regex, ifc_physical_quantity.Name, re.IGNORECASE):
                            add_object = True
                            break
                elif ifc_rel.is_a('IfcRelDefinesByType'):
                    ifc_type_object = ifc_rel.RelatingType
                    if ifc_type_object.Name and re.match(fire_regex, ifc_type_object.Name, re.IGNORECASE):
                        add_object = True
                        break
            if add_object:
                filtered_ifc_objects.append(ifc_object)
        return filtered_ifc_objects

    @staticmethod
    def _has_property_sets(ifc_object, ifc_property_set_names):
        """Helper function to check if an 'IfcObject' contains one of the given 'IfcPropertySets' and
        'IfcElementQuantities' (by name).

                :param ifc_object: 'IfcObject' for checking.
                :param ifc_property_set_names: A list of the names of the 'IfcPropertySets' and 'IfcElementQuantities'.

                :return: 'True' if the given 'IfcObject' contains one of the searched 'IfcPropertySets', otherwise 'False'.
        """
        has_property_sets = False
        for ifc_rel in ifc_object.IsDefinedBy:
            if ifc_rel.is_a('IfcRelDefinesByProperties'):
                ifc_property_set = ifc_rel.RelatingPropertyDefinition
                if ifc_property_set.Name and ifc_property_set.Name in ifc_property_set_names:
                    has_property_sets = True
                    break
        return has_property_sets

    @staticmethod
    def _has_type_objects(ifc_object, ifc_type_object_names):
        """Helper function to check if an 'IfcObject' is typed by one of the given 'IfcTypeObjects' (by name).
```

```
            :param ifc_object: 'IfcObject' for checking.
            :param ifc_type_object_names: A list of the names of the 'IfcTypeObjects'.

            :return: 'True' if the given 'IfcObject' contains one of the searched '
    IfcTypeObjects', otherwise 'False'.
        """
        has_type_objects = False
        for ifc_rel in ifc_object.IsDefinedBy:
            if ifc_rel.is_a('IfcRelDefinesByType'):
                ifc_type_object = ifc_rel.RelatingType
                if ifc_type_object.Name and ifc_type_object.Name in ifc_type_object_names:
                    has_type_objects = True
                    break
        return has_type_objects

    @staticmethod
    def _is_float(value):
        try:
            float(value)
            return True
        except ValueError:
            return False
```

Listing B.5: ifc_prototype.py

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

"""IFC prototype with STEP.

Prototypical implementation of an application to maintain facilities (facility management)
    or managing the ordering
process during construction. This prototype uses a STEP file as base and for serialization.

Note: Many of the optional attributes of IFC elements have been omitted to keep this
    prototype as simple as possible.

"""

from gui.application_gui import *

__author__ = "Martin Klaude"
__status__ = "Prototype"


class StepIfcApplication(IfcApplication):
    def __init__(self):
        super(StepIfcApplication, self).__init__()
        self.setWindowTitle("IFC Prototype")
        self.ifc_file = None
        self.ifc_file_name = None
        self.action_load.triggered.connect(self.load_file)
        self.action_save.triggered.connect(self.save_file)
        self.status_bar.showMessage("Please load an IFC file via the 'File' menu.", 0)

    def load_file(self):
        """Handles the loading of a selected IFC file.

```

```
            :return: ’None’ if the selected IFC file schema does not equal the version ’IFC2X3’.
        """
        self.model.clear()
        self.model.setColumnCount(2)
        self.ifc_file_name, _ = QtWidgets.QFileDialog.getOpenFileName(self, ’Select an IFC file’,
                              ’../ifcr.examples’, ’IFC files (*.ifc)’)
        if self.ifc_file_name:
            self.status_bar.clearMessage()
            self.ifc_file = ifcopenshell.open(self.ifc_file_name)
            if self.ifc_file.schema != "IFC2X3":
                self.status_bar.showMessage("Currently only IFC2X3 is supported!", 5000)
                return
            self.status_bar.showMessage("File ’%s’ opened successfully!" % self.ifc_file_name, 5000)
            self.init_tree_view()

    def save_file(self):
        """Handles the saving of the previously loaded IFC file. """
        if self.ifc_file_name:
            self.ifc_file.write(self.ifc_file_name)
            self.status_bar.showMessage("File ’%s’ saved successfully!" % self.ifc_file_name, 5000)

    def init_tree_view(self):
        """Initializes the tree view by iterating over the ’IfcObjects’. """
        if self.ifc_file is not None:
            # Simply list all ’IfcObjects’ in this prototype.
            for ifc_object in self.ifc_file.by_type(’IfcObject’):
                self.add_ifc_object_item(self.model.invisibleRootItem(), ifc_object)

    def add_ifc_object_item(self, tree_view_root, ifc_object):
        """Adds the given ’IfcObject’ as an item to the given tree view.

                :param tree_view_root: The root of the tree view.
                :param ifc_object: The ’IfcObject’ for adding.
        """
        if ifc_object.Name:
            ifc_object_item = IfcObjectItem(ifc_object.GlobalId, ifc_object.Name)
        else:
            ifc_object_item = IfcObjectItem(ifc_object.GlobalId)
        tree_view_root.appendRow(ifc_object_item)
        if self.action_show_maintenance_status.isChecked():
            super().update_maintenance_status_in_view(ifc_object_item)
        if self.action_show_material_lists.isChecked():
            super().update_material_list_in_view(ifc_object_item)

    def add_new_ifc_property_set(self, ifc_object_global_id, new_ifc_property_set_name):
        """Adds an new ’IfcPropertySet’ to the given ’IfcObject’.

                :param ifc_object_global_id: The ’GlobalId’ of the ’IfcObject’ which should receive the new property set.
                :param new_ifc_property_set_name: The name of the newly created property set.

                :return: The newly created ’IfcPropertySet’.
        """
```

```python
        # An 'IfcPropertySet' will be created with an empty list of properties to keep this prototype as simple as
        #  possible. However, given the IFC specification an 'IfcPropertySet' must have at least one 'IfcProperty'
        #  (otherwise a property set would not make any sense).
        new_ifc_property_set = self.ifc_file.createIfcPropertySet(
            ifcopenshell.guid.compress(uuid.uuid1().hex), self.ifc_file.by_type("IfcOwnerHistory")[0],
            new_ifc_property_set_name, None, [])
        ifc_object = self.ifc_file.by_guid(ifc_object_global_id)
        self.__add_ifc_rel_defines_by_properties(ifc_object, new_ifc_property_set)
        return new_ifc_property_set

    def add_new_ifc_simple_property(self, ifc_property_set, ifc_simple_property_subtype, new_ifc_property_name,
                                                          new_ifc_property_values):
        """Adds an new 'IfcSimpleProperty' to the given 'IfcPropertySet' or 'IfcComplexProperty'.

                :param ifc_property_set: 'IfcPropertySet' or 'IfcComplexProperty' which should receive the new property.
                :param ifc_simple_property_subtype: The subtype of the 'IfcSimpleProperty'.
                :param new_ifc_property_name: The name of the newly created property.
                :param new_ifc_property_values: A dictionary with the new values in the following format: e.g.
                {'NominalValue': 'My test value', 'Unit': ''}.
        """
        new_ifc_property = None
        if ifc_simple_property_subtype == 'IfcPropertySingleValue':
            # Note: Only 'IfcPropertySingleValue' will be handled in this prototype.
            new_ifc_property = self.ifc_file.createIfcPropertySingleValue(new_ifc_property_name, None, None, None)
            new_ifc_property.NominalValue = self.ifc_file.createIfcText(new_ifc_property_values['NominalValue'])
            # Note: 'Unit' has been omitted to keep this prototype as simple as possible.
        if new_ifc_property is not None:
            if ifc_property_set.is_a('IfcPropertySet') and ifc_property_set.DefinesType:
                # Hint: Adding an 'IfcProperty' to an 'IfcPropertySet' of an 'IfcTypeObject' must be handled differently
                #  or be prohibited completely unless the new property is intended to be applied to each occurrence
                #  typed that way.
                # Hint (2): Merging as defined by the IFC and as been done in the IFC2CROI transformation needs
                #  consideration when adding new properties.
                pass
        else:
            ifc_property_set.HasProperties = ifc_property_set.HasProperties + (new_ifc_property,)

    def update_ifc_simple_property(self, ifc_simple_property, new_values):
        """Updates the values of the given 'IfcSimpleProperty'.

                :param ifc_simple_property: The 'IfcSimpleProperty' which should be updated.
                :param new_values: A dictionary with the new values in the following format: e.g.
                {'NominalValue': 'My test value', 'Unit': ''}.
        """
```

```python
        if ifc_simple_property.is_a('IfcPropertySingleValue'):
            new_nominal_value = new_values['NominalValue']
            old_nominal_value = ifc_simple_property.NominalValue
            if old_nominal_value != new_nominal_value:
                ifc_simple_property.NominalValue = self.ifc_file.createIfcText(new_nominal_value)
            # Note: 'Unit' has been omitted to keep this prototype as simple as possible.

    def remove_ifc_property_set(self, ifc_object_global_id, ifc_property_set_global_id):
        """Removes an specific 'IfcPropertySet' from the given 'IfcObject'.

                :param ifc_object_global_id: The 'GlobalId' of the 'IfcObject' which should be updated.
                :param ifc_property_set_global_id: The 'GlobalId' of the 'IfcPropertySet' which should be deleted.
        """
        ifc_object = self.ifc_file.by_guid(ifc_object_global_id)
        for ifc_rel_defines in ifc_object.IsDefinedBy:
            if ifc_rel_defines.is_a('IfcRelDefinesByProperties'):
                relating_property_definition = ifc_rel_defines.RelatingPropertyDefinition
                if relating_property_definition.is_a('IfcPropertySet') \
                        and relating_property_definition.GlobalId == ifc_property_set_global_id:
                    # The order is important here because removing the 'IfcRelDefines' element first would modify the
                    #  reference in memory of the 'RelatingPropertyDefinition' but not the element itself from the
                    #  IFC/STEP file.
                    self.ifc_file.remove(relating_property_definition)
                    self.ifc_file.remove(ifc_rel_defines)
                    break

    def remove_ifc_property(self, ifc_property_set, ifc_property_name):
        """Removes an specific 'IfcProperty' from the given 'IfcPropertySet' or 'IfcComplexProperty'.

                :param ifc_property_set: The 'IfcPropertySet' or 'IfcComplexProperty' which should be updated.
                :param ifc_property_name: The name of the 'IfcProperty' which should be deleted.
        """
        ifc_property = self.find_ifc_property_by_name(ifc_property_set, ifc_property_name)
        if ifc_property is not None:
            self.ifc_file.remove(ifc_property)
            # Remove 'IfcPropertySet' if it has no properties anymore.
            if not ifc_property_set.HasProperties:
                if ifc_property_set.is_a('IfcPropertySet'):
                    self.remove_ifc_property_set(
                        ifc_property_set.PropertyDefinitionOf[0].RelatedObjects[0].GlobalId, ifc_property_set.GlobalId)
                else:
                    # Note: If an 'IfcComplexProperty' has no properties anymore, it must be deleted. This has been
                    #  omitted to keep this prototype as simple as possible.
                    pass

    def find_ifc_property_set_by_name(self, ifc_object_global_id, ifc_property_set_name):
        """Searches for a property set of an 'IfcObject' by name.

```

```python
                :param ifc_object_global_id: The 'GlobalId' of the 'IfcObject' for retrieving the property set.
                :param ifc_property_set_name: The name for the search.

                :return: The property set if found, otherwise 'None'.
        """
        ifc_object = self.ifc_file.by_guid(ifc_object_global_id)
        for ifc_rel_defines in ifc_object.IsDefinedBy:
            if ifc_rel_defines.is_a('IfcRelDefinesByProperties'):
                relating_property_definition = ifc_rel_defines.RelatingPropertyDefinition
                if relating_property_definition.is_a('IfcPropertySet') \
                        and relating_property_definition.Name == ifc_property_set_name:
                    return relating_property_definition
        return None

    @staticmethod
    def find_ifc_property_by_name(ifc_property_set, ifc_property_name):
        """Searches for a property of an 'IfcPropertySet' or 'IfcComplexProperty' by name.

                :param ifc_property_set: The 'IfcPropertySet' or 'IfcComplexProperty' for retrieving the property.
                :param ifc_property_name: The name for the search.

                :return: The property if found, otherwise 'None'.
        """
        for ifc_property in ifc_property_set.HasProperties:
            if ifc_property.Name == ifc_property_name:
                return ifc_property
        return None

    @staticmethod
    def get_material_list(material_list_property_set):
        """Helper function to retrieve all properties of the 'MaterialList' property set in a dictionary in the
        following format: e.g. {'Nails': '42', 'Wooden Planks': '3'}.

                :param material_list_property_set: The corresponding property set containing all information.

                :return: A dictionary containing the material as key and the count as value.
        """
        material_list = {}
        for material_property in material_list_property_set.HasProperties:
            if material_property.is_a('IfcPropertySingleValue'):
                material_list[material_property.Name] = material_property.NominalValue.wrappedValue
        return material_list

    def get_maintenance_status(self, maintenance_property_set):
        """Helper function to retrieve all maintenance information of the given property set in a dictionary in the
        following format: e.g. {'Status': 'Broken', 'Information': 'Door handle broken'}.

                :param maintenance_property_set: The corresponding property set containing all information.

                :return: A dictionary containing all maintenance information (status and info).
```

```
        """
        maintenance_status = self.find_ifc_property_by_name(maintenance_property_set, 'Status')
        if maintenance_status is not None and maintenance_status.is_a('IfcPropertySingleValue'):
            maintenance_status = maintenance_status.NominalValue.wrappedValue
        maintenance_information = self.find_ifc_property_by_name(maintenance_property_set, 'Information')
        if maintenance_information is not None and maintenance_information.is_a('IfcPropertySingleValue'):
            maintenance_information = maintenance_information.NominalValue.wrappedValue
        return {'Status': maintenance_status, 'Information': maintenance_information}

    def __add_ifc_rel_defines_by_properties(self, ifc_object, ifc_property_set):
        """Connects the given 'IfcObject' and 'IfcPropertySet' by creating the necessary 'IfcRelDefinesByProperties'
        relationship.

                :param ifc_object: The related 'IfcObject'.
                :param ifc_property_set: The relating 'IfcPropertySet'.
        """
        new_ifc_rel_defines_by_properties = self.ifc_file.createIfcRelDefinesByProperties(
            ifcopenshell.guid.compress(uuid.uuid1().hex), self.ifc_file.by_type("IfcOwnerHistory")[0])
        new_ifc_rel_defines_by_properties.RelatedObjects = [ifc_object]
        new_ifc_rel_defines_by_properties.RelatingPropertyDefinition = ifc_property_set


def exception_hook(exception_class, exception_instance, exception_traceback):
    """Defines an exception hook for the GUI application.

            :param exception_class: The exception class.
            :param exception_instance: The exception instance.
            :param exception_traceback: The exception traceback.
    """
    error_traceback = "".join(traceback.format_exception(exception_class, exception_instance, exception_traceback))
    print("An error occurred while running the IFC/STEP application! Error: \n\t%s" % error_traceback)
    QtWidgets.QApplication.quit()


def main():
    sys.excepthook = exception_hook
    step_application = QtWidgets.QApplication(sys.argv)
    main_window = StepIfcApplication()
    main_window.show()
    status_code = step_application.exec_()
    sys.exit(status_code)


if __name__ == '__main__':
    main()
```

Listing B.6: ifcr_prototype.py

```python
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

"""IFC prototype with CROM/CROI.

Prototypical implementation of an application to maintain facilities (facility management) or managing the ordering
process during construction. This prototype uses a CROM/CROI (role-oriented approach) as base and for serialization.

Note (1): The CROM needs to be in the same directory as the loaded CROI.
Note (2): Many of the optional attributes of IFC elements have been omitted to keep this prototype as simple as
possible.

"""

import glob
import re

from gui.application_gui import *
from pyecore.resources import ResourceSet, URI
from pyecore.utils import DynamicEPackage
from xml.etree import ElementTree

__author__ = "Martin Klaude"
__status__ = "Prototype"

CROM_CROI_DIALOG_UI, _ = uic.loadUiType('gui/crom_croi_dialog.ui')


class RolesIfcApplication(IfcApplication):
    def __init__(self):
        super(RolesIfcApplication, self).__init__()
        self.setWindowTitle("IFC-R Prototype")
        self.working_dir = None
        self.crom = None
        self.crom_file = None
        self.crom_file_name = None
        self.croi = None
        self.croi_file = None
        self.croi_file_name = None
        self.action_load.triggered.connect(self.load_working_dir)
        self.action_save.triggered.connect(self.save_files)
        self.status_bar.showMessage(
            "Please load the working directory containing the CROM and CROI via the 'File' menu.", 0)

    def load_working_dir(self):
        """Handles the loading of a selected working directory. This directory must hold the CROM and the CROI. """
        self.model.clear()
        self.model.setColumnCount(2)
        self.working_dir = QtWidgets.QFileDialog.getExistingDirectory(
            self, 'Select working directory', '../',
            QtWidgets.QFileDialog.ShowDirsOnly | QtWidgets.QFileDialog.DontResolveSymlinks)
        if self.working_dir:
```

```
        crom_croi_dialog = CromCroiDialog(self.working_dir)
        dialog_status = crom_croi_dialog.exec_()
        if dialog_status == QtWidgets.QDialog.Accepted and crom_croi_dialog.crom_file_name is not None \
                and crom_croi_dialog.croi_file_name is not None:
            # TODO: A check needs to be implemented in a real implementation that an user has selected matching
            #  CROM and CROI files.
            self.crom_file_name = crom_croi_dialog.crom_file_name
            self.croi_file_name = crom_croi_dialog.croi_file_name
            self.__load_crom()
            self.croi_file = ElementTree.parse(self.croi_file_name)
            self.croi = self.croi_file.getroot()
            self.status_bar.clearMessage()
            self.status_bar.showMessage(
                "CROM ('%s') / CROI ('%s') opened successfully!" % (self.crom_file_name, self.croi_file_name), 5000)
            self.init_tree_view()

def save_files(self):
    """Handles the saving of the previously loaded CROM and CROI files. """
    if self.croi_file_name and self.crom_file_name:
        self.croi_file.write(self.croi_file_name, 'UTF-8', True)
        self.crom_file.save()
        self.status_bar.showMessage(
            "CROM ('%s') / CROI ('%s') saved successfully!" % (self.crom_file_name, self.croi_file_name), 5000)

def init_tree_view(self):
    """Initializes the tree view by iterating over the 'IfcObjects'. """
    if self.crom is not None and self.croi is not None:
        # Add view information to the tree view first.
        if self.croi.find('viewInformation'):
            for view_information in self.croi.find('viewInformation'):
                view_information_item = QtGui.QStandardItem('View Information')
                self.model.invisibleRootItem().appendRow(view_information_item)
                if self.__is_in_crom(view_information, 'DataType'):
                    view_information_item.appendRow([QtGui.QStandardItem(view_information.tag[3:]),
                                                     QtGui.QStandardItem(view_information.attrib['value'])])
        for ifc_object in self.croi.find('ifcObjects'):
            # Checking if the 'IfcObject' (CROI element) is part of the CROM because model and metamodel need to be
            #  consistent. This has been implemented because this prototype does not reflect a 'real' role-oriented
            #  approach where this check would be obsolete.
            if self.__is_in_crom(ifc_object, 'NaturalType'):
                self.add_ifc_object_item(self.model.invisibleRootItem(), ifc_object)

def add_ifc_object_item(self, tree_view_root, ifc_object):
    """Adds the given 'IfcObject' as an item to the given tree view.

            :param tree_view_root: The root of the tree view.
            :param ifc_object: The 'IfcObject' for adding.
    """
    if 'name' in ifc_object.attrib:
        ifc_object_item = IfcObjectItem(ifc_object.attrib['globalId'], ifc_object.attrib['name'])
```

```python
        else:
            ifc_object_item = IfcObjectItem(ifc_object.attrib['globalId'])
        tree_view_root.appendRow(ifc_object_item)
        if self.action_show_maintenance_status.isChecked():
            super().update_maintenance_status_in_view(ifc_object_item)
        if self.action_show_material_lists.isChecked():
            super().update_material_list_in_view(ifc_object_item)

    def add_new_ifc_property_set(self, ifc_object_global_id, new_ifc_property_set_name):
        """Adds an new 'IfcPropertySet' to the given 'IfcObject'.

                :param ifc_object_global_id: The 'GlobalId' of the 'IfcObject' which should receive the new property set.
                :param new_ifc_property_set_name: The name of the newly created property set.

                :return: The newly created 'IfcPropertySet'.
        """
        # This function reflects the adding/playing of a new role if compared to the 'Role Object Pattern'.
        specification_name = 'PS_%s' % re.sub(r'[^a-zA-Z]', '', new_ifc_property_set_name)
        new_ifc_property_set = ElementTree.Element(specification_name)
        new_ifc_property_set.attrib['name'] = new_ifc_property_set_name
        new_ifc_property_set.attrib['globalId'] = ifcopenshell.guid.compress(uuid.uuid1().hex)
        if not self.__is_in_crom(new_ifc_property_set, 'RoleType'):
            # Adds the 'IfcPropertySet' role to the CROM because the metamodel needs to be consistent with the
            #  corresponding model (CROI).
            self.__add_ifc_property_set_role_to_crom(specification_name)
        object_definition = self.__find_object_definition_for_player(ifc_object_global_id)
        if object_definition is None:
            object_definition = self.__create_new_object_definition(ifc_object_global_id)
        object_definition.find('plays').append(new_ifc_property_set)
        return new_ifc_property_set

    def add_new_ifc_simple_property(self, ifc_property_set, ifc_simple_property_subtype, new_ifc_property_name,
                                                                  new_ifc_property_values):
        """Adds an new 'IfcSimpleProperty' to the given 'IfcPropertySet' or 'IfcComplexProperty'.

                :param ifc_property_set: 'IfcPropertySet' or 'IfcComplexProperty' which should receive the new property.
                :param ifc_simple_property_subtype: The subtype of the 'IfcSimpleProperty'.
                :param new_ifc_property_name: The name of the newly created property.
                :param new_ifc_property_values: A dictionary with the new values in the following format: e.g.
                {'NominalValue': 'My test value', 'Unit': ''}.
        """
        new_ifc_property = ElementTree.Element(ifc_simple_property_subtype)
        if self.__is_in_crom(new_ifc_property, 'DataType'):
            new_ifc_property.attrib['name'] = new_ifc_property_name
            self.__update_ifc_property(new_ifc_property, new_ifc_property_values)
            ifc_property_set.append(new_ifc_property)

    def update_ifc_simple_property(self, ifc_simple_property, new_values):
        """Updates the values of the given 'IfcSimpleProperty'.

```

```
            :param ifc_simple_property: The 'IfcSimpleProperty' which should be updated.
            :param new_values: A dictionary with the new values in the following format: e.g.
            {'NominalValue': 'My test value', 'Unit': ''}.
        """
        self.__update_ifc_property(ifc_simple_property, new_values)

    def remove_ifc_property_set(self, ifc_object_global_id, ifc_property_set_global_id):
        """Removes an specific 'IfcPropertySet' from the given 'IfcObject'.

            :param ifc_object_global_id: The 'GlobalId' of the 'IfcObject' which should be updated.
            :param ifc_property_set_global_id: The 'GlobalId' of the 'IfcPropertySet' which should be deleted.
        """
        # This function reflects the dropping of a role if compared to the 'Role Object Pattern'.
        object_definition = self.__find_object_definition_for_player(ifc_object_global_id)
        if object_definition is not None:
            for played_role in object_definition.find('plays'):
                if played_role.attrib['globalId'] == ifc_property_set_global_id:
                    roles = object_definition.find('plays')
                    roles.remove(played_role)
                    # Remove from CROM if not played by any player because roles are not rigid (or anti-rigid).
                    if not self.__is_played(played_role.tag):
                        self.__remove_role_from_crom(played_role.tag)
                    if not roles:
                        # Cleanup 'Object Definition' if it has no roles anymore.
                        self.croi.find('objectDefinitions').remove(object_definition)

    def remove_ifc_property(self, ifc_property_set, ifc_property_name):
        """Removes an specific 'IfcProperty' from the given 'IfcPropertySet' or 'IfcComplexProperty'.

            :param ifc_property_set: The 'IfcPropertySet' or 'IfcComplexProperty' which should be updated.
            :param ifc_property_name: The name of the 'IfcProperty' which should be deleted.
        """
        ifc_property = self.find_ifc_property_by_name(ifc_property_set, ifc_property_name)
        if ifc_property is not None:
            ifc_property_set.remove(ifc_property)
            # Remove 'IfcPropertySet' if it has no properties anymore.
            if not ifc_property_set:
                if ifc_property_set.tag.startswith('PS_'):
                    object_definition = self.__find_object_definition_for_role(ifc_property_set.attrib['globalId'])
                    if object_definition is not None:
                        self.remove_ifc_property_set(
                        object_definition.find('player').attrib['referenceId'], ifc_property_set.attrib['globalId'])
                else:
                    # Note: If an 'IfcComplexProperty' has no properties anymore, it must be deleted. This has been
                    #  omitted to keep this prototype as simple as possible.
                    pass

```

```
    def find_ifc_property_set_by_name(self, ifc_object_global_id, ifc_property_set_name):
        """Searches for a property set of an 'IfcObject' by name.

            :param ifc_object_global_id: The 'GlobalId' of the 'IfcObject' for retrieving the property set.
            :param ifc_property_set_name: The name for the search.

            :return: The property set if found, otherwise 'None'.
        """
        # This function reflects the retrieval of a played role of a core if compared to the 'Role Object Pattern'.
        for object_definition in self.croi.find('objectDefinitions'):
            if object_definition.find('player').attrib['referenceId'] == ifc_object_global_id:
                for played_role in object_definition.find('plays'):
                    if played_role.tag.startswith('PS_') and played_role.attrib['name'] == ifc_property_set_name:
                        return played_role
        return None

    @staticmethod
    def find_ifc_property_by_name(ifc_property_set, ifc_property_name):
        """Searches for a property of an 'IfcPropertySet' by name.

            :param ifc_property_set: The 'IfcPropertySet' for retrieving the property.
            :param ifc_property_name: The name for the search.

            :return: The property if found, otherwise 'None'.
        """
        for ifc_property in ifc_property_set:
            if ifc_property.attrib['name'] == ifc_property_name:
                return ifc_property
        return None

    @staticmethod
    def get_material_list(material_list_property_set):
        """Helper function to retrieve all properties of the 'MaterialList' property set in a dictionary in the
        following format: e.g. {'Nails': '42', 'Wooden Planks': '3'}.

            :param material_list_property_set: The corresponding property set containing all information.

            :return: A dictionary containing the material as key and the count as value.
        """
        material_list = {}
        for material_property in material_list_property_set:
            if material_property.tag.startswith('IfcPropertySingleValue'):
                material_list[material_property.attrib['name']] = material_property.attrib['nominalValue']
        return material_list

    def get_maintenance_status(self, maintenance_property_set):
        """Helper function to retrieve all maintenance information of the given property set in a dictionary in the
        following format: e.g. {'Status': 'Broken', 'Information': 'Door handle broken'}.

            :param maintenance_property_set: The corresponding property set containing
```

```
all information.

            :return: A dictionary containing all maintenance information (status and info).
        """
        maintenance_status = self.find_ifc_property_by_name(maintenance_property_set, 'Status')
        if maintenance_status is not None and maintenance_status.tag.startswith('IfcPropertySingleValue'):
            maintenance_status = maintenance_status.attrib['nominalValue']
        maintenance_information = self.find_ifc_property_by_name(maintenance_property_set, 'Information')
        if maintenance_information is not None and maintenance_information.tag.startswith('IfcPropertySingleValue'):
            maintenance_information = maintenance_information.attrib['nominalValue']
        return {'Status': maintenance_status, 'Information': maintenance_information}

    def __load_crom(self):
        """Loads the CROM Ecore as meta model by using the PyEcore package. """
        crom_rset = ResourceSet()
        crom_mm_resource = crom_rset.get_resource(URI('../ifcr.metamodels/CROM.ecore'))
        crom_mm_root = crom_mm_resource.contents[0]
        crom_rset.metamodel_registry[crom_mm_root.nsURI] = crom_mm_root
        # Create all elements (EClasses, etc.) from the CROM meta model (Ecore).
        self.crom = DynamicEPackage(crom_mm_root)
        self.crom_file = crom_rset.get_resource(URI(self.crom_file_name))

    def __is_in_crom(self, croi_element, crom_element):
        """Checks if the given CROI element is part of the CROM.

            :param croi_element: The CROI element.
            :param crom_element: The corresponding CROM element, e.g. 'NaturalType'.

            :return: 'True' if the CROI element is part of the CROM, otherwise 'False'.
        """
        if crom_element == 'NaturalType':
            return any(nt.name == croi_element.tag for nt in self.crom.NaturalType.allInstances())
        if crom_element == 'RoleType':
            return any(rt.name == croi_element.tag for rt in self.crom.RoleType.allInstances())
        if crom_element == 'DataType':
            return any(dt.name == croi_element.tag for dt in self.crom.DataType.allInstances())

    def __is_played(self, role_tag):
        """Checks if any player plays the given role by tag id.

            :param role_tag: The CROI element tag of the role.

            :return: 'True' if any player plays the given role, otherwise 'False'.
        """
        for object_definition in self.croi.find('objectDefinitions'):
            for played_role in object_definition.find('plays'):
                if played_role.tag == role_tag:
                    return True
        return False

    def __find_object_definition_for_player(self, ifc_global_id):
```

```
        """Searches for an 'Object Definition' of the given 'IfcObject' via 'GlobalId'.

                :param ifc_global_id: The 'GlobalId' of the 'IfcObject'.

                :return: The corresponding 'Object Definition' if found, otherwise 'None'.
        """
        # This function reflects the retrieval of the context ('Compartment') for a given object (player).
        for object_definition in self.croi.find('objectDefinitions'):
            if object_definition.find('player').attrib['referenceId'] == ifc_global_id:
                return object_definition
        return None

    def __find_object_definition_for_role(self, ifc_global_id):
        """Searches for an 'Object Definition' of the given 'IfcPropertySet', 'IfcElementQuantity' or 'IfcTypeObject'
        role via 'GlobalId'.

                :param ifc_global_id: The 'GlobalId' of the 'IfcPropertySet', 'IfcElementQuantity' or 'IfcTypeObject'
                role.

                :return: The corresponding 'Object Definition' if found, otherwise 'None'.
        """
        # This function reflects the retrieval of the core of a played role if compared to the 'Role Object Pattern'.
        for object_definition in self.croi.find('objectDefinitions'):
            for played_role in object_definition.find('plays'):
                if played_role.attrib['globalId'] == ifc_global_id:
                    return object_definition
        return None

    def __add_ifc_property_set_role_to_crom(self, role_name):
        """Adds an 'IfcPropertySet' role to the CROM.

                :param role_name: The name of the newly added 'IfcPropertySet' role.
        """
        crom_object_definition = [ct for ct in self.crom.CompartmentType.allInstances() if
                                            ct.name == 'ObjectDefinition'][0]
        new_ifc_property_set_role = self.crom.RoleType()
        new_ifc_property_set_role.name = role_name
        crom_compartment_part = self.crom.Part()
        crom_compartment_part.role = new_ifc_property_set_role
        crom_object_definition.parts.append(crom_compartment_part)
        crom_role_inheritance = self.crom.RoleInheritance()
        crom_role_inheritance.super = [rt for rt in self.crom.RoleType.allInstances() if
                                                     rt.name == 'IfcPropertySet'][0]
        crom_role_inheritance.sub = new_ifc_property_set_role
        list(self.crom.Model.allInstances())[0].relations.append(crom_role_inheritance)

    def __remove_role_from_crom(self, role_name):
        """Removes a role and its role inheritance from the CROM by name.

                :param role_name: The name of the role which should be removed.
        """
        crom_object_definition = [ct for ct in self.crom.CompartmentType.allInstances() if
                                            ct.name == 'ObjectDefinition'][0]
        ifc_property_set_role_part = [part for part in self.crom.Part.allInstances() if
```

```
    part.role.name == role_name][0]
        crom_object_definition.parts.remove(ifc_property_set_role_part)
        crom_role_inheritance = [ri for ri in self.crom.RoleInheritance.allInstances() if
                                                  ri.sub == ifc_property_set_role_part.role
    ][0]
        list(self.crom.Model.allInstances())[0].relations.remove(crom_role_inheritance)

    def __create_new_object_definition(self, player_reference_id):
        """Creates an new 'ObjectDefinition'.

                :param player_reference_id: The 'GlobalId' of the 'IfcObject' player.

                :return: The newly created 'ObjectDefinition' in the CROI for further
    processing.
        """
        new_object_definition = ElementTree.Element('objectDefinition')
        new_object_definition.attrib['name'] = 'od%s' % (len(self.croi.find('
    objectDefinitions')) + 1)
        new_object_definition_player = ElementTree.Element('player')
        new_object_definition_player.attrib['referenceId'] = player_reference_id
        new_object_definition.append(new_object_definition_player)
        new_object_definition_plays = ElementTree.Element('plays')
        new_object_definition.append(new_object_definition_plays)
        self.croi.find('objectDefinitions').append(new_object_definition)
        return new_object_definition

    def __update_ifc_property(self, ifc_property, new_ifc_property_values):
        """Updates/Adds values of the given 'IfcProperty'.

                :param ifc_property: The 'IfcProperty' to update.
                :param new_ifc_property_values: A dictionary with the new values in the
    following format: e.g.
                {'NominalValue': 'My test value', 'Unit': ''}.
        """
        crom_data_type = [dt for dt in self.crom.DataType.allInstances() if dt.name ==
    ifc_property.tag][0]
        for attribute in crom_data_type.attributes:
            new_value = new_ifc_property_values[attribute.name[0].upper() + attribute.name
    [1:]]
            if attribute.name in ifc_property.attrib:
                old_value = ifc_property.attrib[attribute.name]
                if old_value != new_value:
                    ifc_property.attrib[attribute.name] = new_value
            else:
                ifc_property.attrib[attribute.name] = new_value


class CromCroiDialog(QtWidgets.QDialog, CROM_CROI_DIALOG_UI):
    def __init__(self, working_dir):
        QtWidgets.QDialog.__init__(self)
        CROM_CROI_DIALOG_UI.__init__(self)
        self.setupUi(self)
        os.chdir(working_dir)
        self.crom_file_name = None
        self.croi_file_name = None
        self.button_box.button(QtWidgets.QDialogButtonBox.Ok).clicked.connect(self.
    select_crom_croi)
        self.__init_crom_combo_box()
        self.__init_croi_combo_box()
```

```python

    def select_crom_croi(self):
        """Saves the selected CROM and CROI. """
        self.crom_file_name = self.crom_combo_box.currentText()
        self.croi_file_name = self.croi_combo_box.currentText()

    def __init_crom_combo_box(self):
        """Initializes the CROM combo box according to the selected working directory. """
        crom_files = glob.glob('*.crom')
        for crom_file in crom_files:
            self.crom_combo_box.addItem(crom_file)

    def __init_croi_combo_box(self):
        """Initializes the CROI combo box according to the selected working directory. """
        croi_files = glob.glob('*.croi')
        for croi_file in croi_files:
            self.croi_combo_box.addItem(croi_file)


def exception_hook(exception_class, exception_instance, exception_traceback):
    """Defines an exception hook for the GUI application.

            :param exception_class: The exception class.
            :param exception_instance: The exception instance.
            :param exception_traceback: The exception traceback.
    """
    error_traceback = "".join(traceback.format_exception(exception_class, exception_instance, exception_traceback))
    print("An error occurred while running the CROM/CROI application! Error: \n\t%s" % error_traceback)
    QtWidgets.QApplication.quit()


def main():
    sys.excepthook = exception_hook
    roles_application = QtWidgets.QApplication(sys.argv)
    main_window = RolesIfcApplication()
    main_window.show()
    status_code = roles_application.exec_()
    sys.exit(status_code)


if __name__ == '__main__':
    main()
```

Listing B.7: ifc_cfc_read_property.py

```python
import ifcopenshell

def main():
    ifc_file = ifcopenshell.open("../../ifcr.examples/simple_house.ifc")
    ifc_door = ifc_file.by_type("IfcDoor")[0]
    for ifc_rel_defines in ifc_door.IsDefinedBy:
        if ifc_rel_defines.is_a("IfcRelDefinesByProperties"):
            ifc_property_set = ifc_rel_defines.RelatingPropertyDefinition
            ifc_property = ifc_property_set.HasProperties[0]
            if ifc_property.is_a("IfcPropertySingleValue"):
                print("First found property:", ifc_property.Name)
                exit()
```

```python

if __name__ == "__main__":
    main()
```

Listing B.8: ifcr_cfc_read_property.py

```python
from xml.etree import ElementTree

def main():
    croi_file = ElementTree.parse("../../ifcr.examples/simple_house.croi")
    croi = croi_file.getroot()
    for ifc_object in croi.find("ifcObjects"):
        if ifc_object.tag == "IfcDoor":
            ifc_door = ifc_object
            for object_definition in croi.find("objectDefinitions"):
                if object_definition.find("player").attrib["referenceId"] == ifc_door.attrib["globalId"]:
                    for ifc_property_set in object_definition.find("plays"):
                        if ifc_property_set.tag.startswith("PS_"):
                            ifc_property = ifc_property_set.find("IfcPropertySingleValue")
                            print("First found property:", ifc_property.attrib["name"])
                            exit()

if __name__ == "__main__":
    main()
```

Listing B.9: ifc_cfc_find_objects.py

```python
import ifcopenshell

def main():
    ifc_file = ifcopenshell.open("../../ifcr.examples/simple_house.ifc")
    ifc_objects = ifc_file.by_type("IfcObject")
    for ifc_object in ifc_objects:
        if ifc_object.IsDefinedBy:
            for ifc_rel_defines in ifc_object.IsDefinedBy:
                if ifc_rel_defines.is_a("IfcRelDefinesByProperties"):
                    ifc_property_set = ifc_rel_defines.RelatingPropertyDefinition
                    if ifc_property_set.Name == "Pset_FireRatingProperties":
                        print(ifc_object.GlobalId)

if __name__ == "__main__":
    main()
```

Listing B.10: ifcr_cfc_find_objects.py

```python
from xml.etree import ElementTree

def main():
    croi_file = ElementTree.parse("../../ifcr.examples/simple_house.croi")
    croi = croi_file.getroot()
    for object_definition in croi.find("objectDefinitions"):
        for ifc_property_set in object_definition.find("plays"):
            if ifc_property_set.tag.startswith("PS_") and ifc_property_set.attrib["name"] == "Pset_FireRatingProperties":
                print(object_definition.find("player").attrib["referenceId"])

if __name__ == "__main__":
    main()
```