Summer Fellowship Report

On

Development of IFC Exporter for Osdag

Submitted by

Mukunth A G

Under the guidance of

Prof. Siddhartha Ghosh
Civil Engineering Department
IIT Bombay

Under the mentorship of

Mr. Danish Ansari
Osdag FOSSEE, IIT Bombay

August 22, 2022

Acknowledgment

I would like to thank FOSSEE Team for providing me with this opportunity. The internship helped me gain knowledge of CAD creation's fundamentals and some practical programming experience.

I thank Prof. Siddhartha Ghosh for giving me this opportunity. I would also like to thank my mentor and guide Mr. Danish Ansari for his kind support in the completion of this project.

Contents

Chapter 1

Introduction

1.1 About Osdag

Osdag is Free/Libre and Open-Source Software being developed for the design of steel structures following IS 800:2007 and other relevant design codes. OSDAG helps users in designing steel connections, members, and systems using an interactive Graphical User Interface (GUI). The source code is written in Python, and 3D CAD images are developed using PythonOCC.

1.2 What is IFC?

IFC (Industry Foundation Class) is a CAD data exchange file format that is intended for the description of building and construction data. It is an object-based file format with the data model developed by buildingSMART to facilitate interoperability in engineering and construction. It is registered by ISO and is an official International Standard ISO 16739-1:2018. There are newer versions of IFC, but currently, IFC2X3 is the most stable. See [1, 2] for more information.

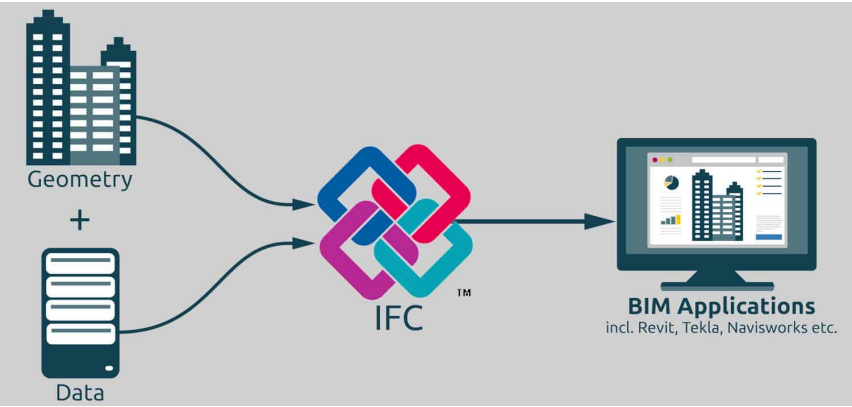

Figure 1: Primary Application of IFC

1.3 Need for IFC in Osdag

Among other CAD file formats such as STEP, IGES, STL, and BREP, IFC stands out as it can hold information associated with specific CAD geometry. IFCs have their own classification of construction objects such as Beam, Column, Plate, MechanicalFastener, etc. This makes it easier to read the properties of each construction element.

In Osdag, since the calculated values of element properties must be associated with the element, IFCs act as a brilliant format. Generally, whereas in formats like STEP, only the CAD geometry can be represented not the properties of each element. IFCs are also a very popular format in BIM Industry which makes it easier for interoperability.

Since the IFCs associate textual information with the CAD objects, a dedicated viewer is required. It turns out that choosing the right viewer is important for the effective readability of properties. Appendix 7.1 covers the choice of IFC viewers.

1.4 The basic structure of an IFC file

Apart from dedicated IFC viewers, an IFC file can also be viewed in any text editor. This feature will be very helpful in troubleshooting IFC files. One such simple file is provided here [3]. One can open the same file in any one of the IFC viewers and observe the relationship between the textual view and the one shown in the GUI of the viewer.

The basic structure of an IFC file is shown in Fig 2. IFC files create a building model based on a pre-defined structure that builds the model in a logical way. When it is saved, the IFC file format orders the IFC units hierarchically according to their type.

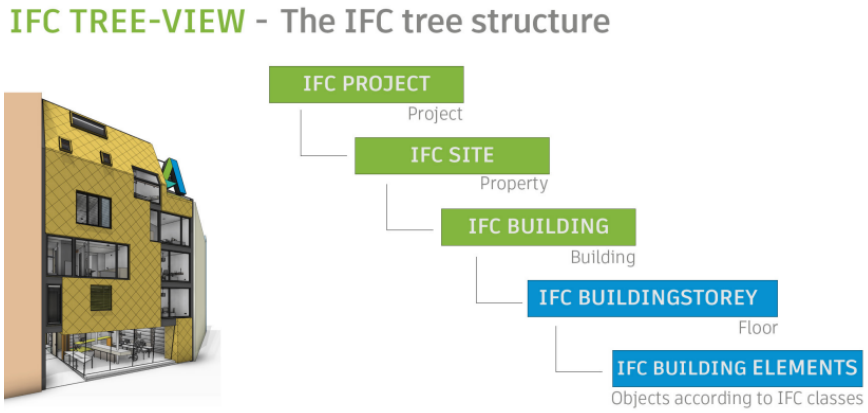

Figure 2: The basic structure of an IFC file [4]

IFC geometry is explained in greater detail in Section 3. The details about each

IFC element can be found here [5]. A simple web search also makes this process faster and is required throughout the development of the IFC exporter.

Chapter 2

Introduction to Osdag's geometry creation

Understanding Osdag's geometry creation is a necessary precursor to beginning the development of IFC exporter. PythonOCC is the main library to form all the shapes in Osdag. But it does not play a major role in building an IFC exporter. Hence we don't discuss it in-depth here. We will first discuss the file/module structure and then move on to the procedure of cad creation in Osdag.

2.1 File Structure

Osdag has main modules and sub-modules. This can be easily seen from the Osdag GUI.

In Fig 3, the left column shows the main modules and the top row covers the sub-modules, further under each module other sub-modules may be present. Generally, the last child module is the module that defines the connectivity type. Fig 4 shows the connectivity modules under the Cleat Angle module.

The geometry creation is completely taken care of inside the directory called "cad", whereas the design data required to form the geometry comes from the "design_type" directory. From here, our main focus is on this "cad" directory and we will be referring to it as just cad.

Inside cad, the "items" directory contains cad items like Bolt, Nut, and ISection, these will be used by other files inside cad for the creation of its geometries. Note: Each file inside items has its own class named the same as the filename (with pascal casing applied), and can also be run as the main file. One can try running the "bolt.py" file to see the bolt created and displayed.

A file where the full geometry is assembled will be referred to as the "main cad file". For example, in the case of Cleat Angle, with Column-Flange-Beam Web Connectivity, a file named colFlangeBeamWebConnectivity.py under "CleatAngle" which is under "Shear Connection" is the main cad file for its creation.

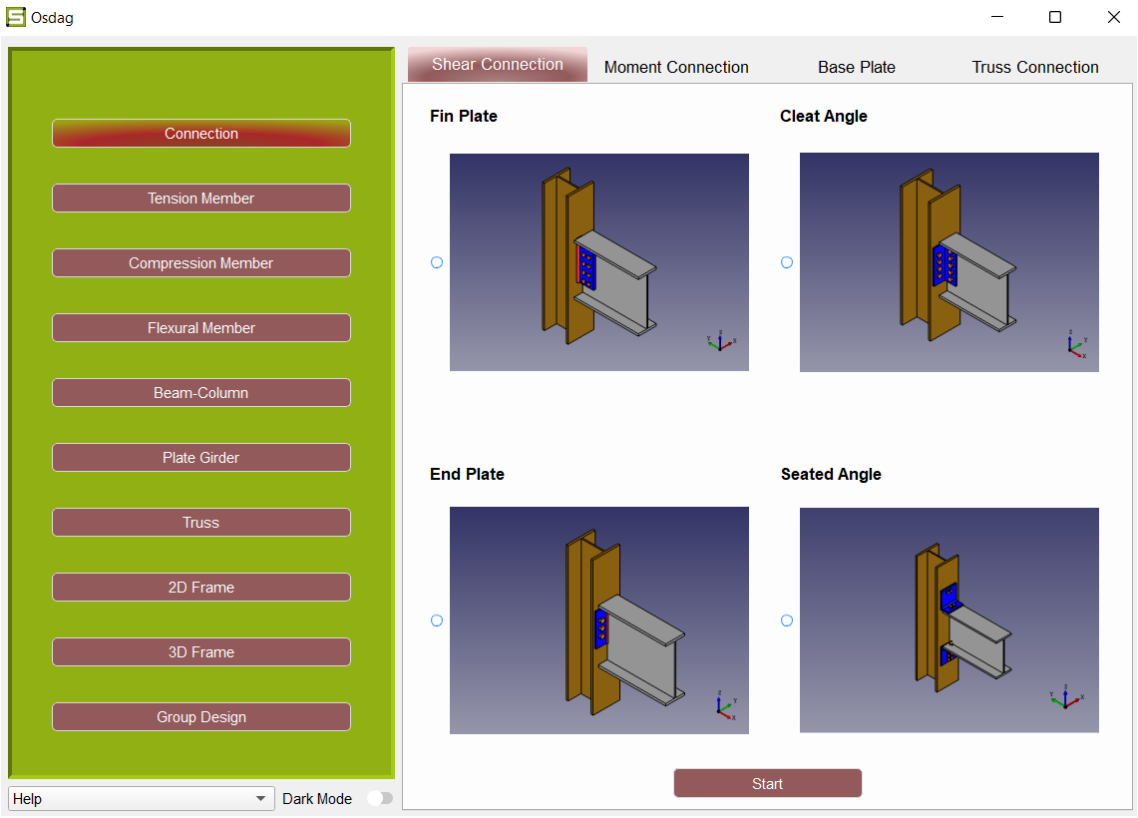

Figure 3: Home window of osdag

The cad items for the assembly are formed in the common_logic.py in CommonDesignLogic class, its respective python method (for Column-Flange-Beam Web connectivity, it is create3DcolFlangeBeamWeb()) which is also inside the cad directory.

2.2 Steps for producing a geometry

Suppose a user selects Cleat Angle with Column-Flange-Beam Web Connectivity and clicks "design". The following functions take place,

A file called ui_template.py in the "gui" directory captures all the gui information (like the selected modules, input values, etc.), and initializes an object of the CommonDesignLogic class which is in common_logic.py. Then, its method call_3DModel() is executed (see Fig 5).

In the call_3DModel() method, we have the connectivity obtained from the module class and the corresponding function is run depending on the connectivity (see Fig 6).

Now the method create3DColFlangeBeamWeb() [6] (it will be referred to as "connectivity function") is executed where the cad items required are created one by one (cad items will be discussed at the end of this section), getting the design

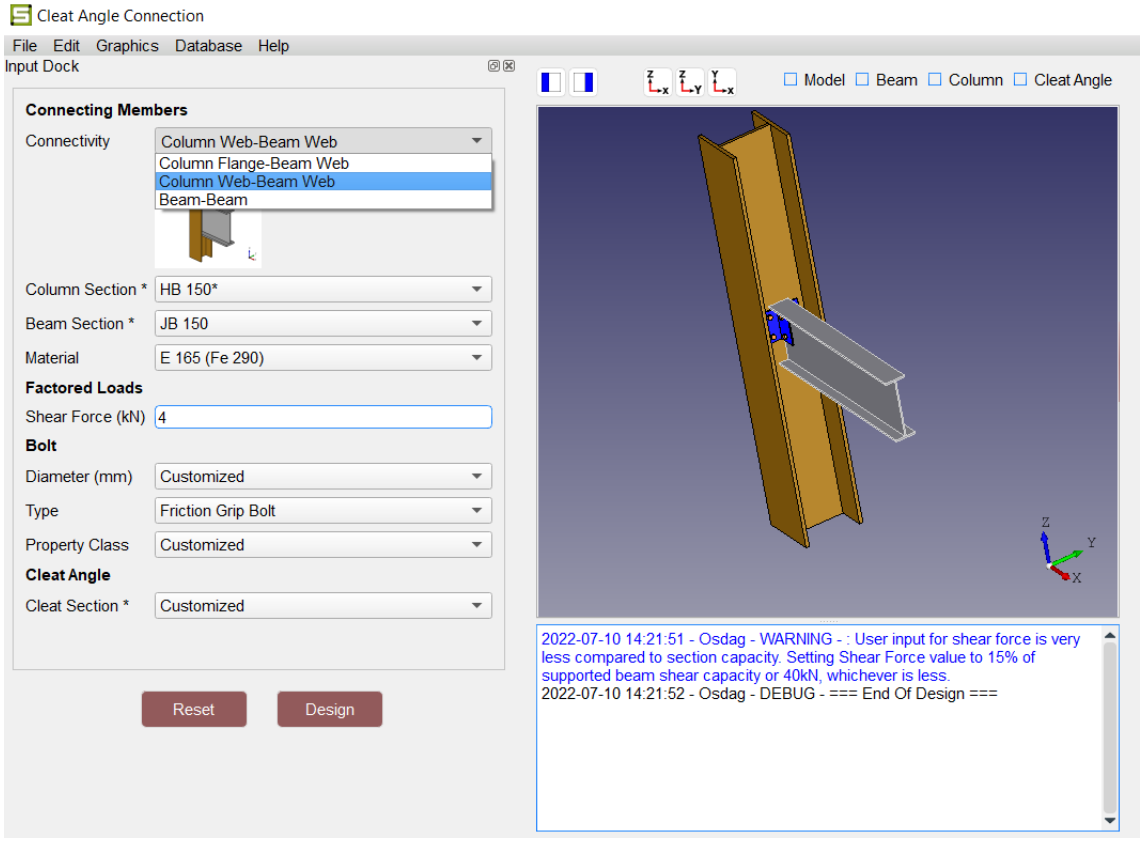

Figure 4: Connectivity modules of Cleat Angle Connection (see left side)

data from the "module_class" object. In Osdag, it is named "A". This object contains all design data required for the creation of cad items, except the exact location and placement, but some location data is derivable from the design data. To understand this better one has to refer to [6].

Now, the main cad file/class for assembling is called which is under the name "cleatColFlangeBeamWeb" in the connectivity function (see Fig 7).

The ColFlangeBeamWeb class in the main cad file gets the necessary items and initializes them to self. It contains methods to create the complete models of all the items by placing them in calculated locations. Now, the "create_3dmodel" method is called to accomplish the same. (see Fig 8) Hence the final model is created.

2.3 Cad Items

Inside the "items" directory of cad, we can find the basic cad items required to complete any geometry. These cad items are generally created in common_logic.py inside the corresponding connectivity function. Here's an example of an Angle getting created in our previous example.

```
print("common start")
self.commLogicObj = CommonDesignLogic(self.display, self.folder, main.module, main.mainmodule)
print("common start")
status = main.design_status
###############trial##############
# status = True
###############trial##############

module_class = self.return_class(main.module)
# self.progress_bar.setValue(80)
print("3D start")
self.commLogicObj.call_3DModel(status, module_class)
print("3D end")
```

Figure 5: Code for creating the "CommonDesignLogic" object in "ui_template.py"

Now, if inside the "Angle" class of angle.py, we can notice that there are three methods, see Fig 10.

- place() - assigns reference directions and calls to compute_params
- compute_params() - creates profile points
- create_model() - creates profile and extrudes the profile (using PythonOCC)

A detailed description of the above methods is not required here. What's important is the return value of create_model(). In the __main__code of angle.py, we have the variable called "prism". If we print the type of this object, we would get a pythonOCC TopoDS class, such as, "class¡'TopoDS_Compound'; id:1002715265¿". This class is then used to display the object.

2.4 Placement of geometry

The example of beam geometry is taken here. The beam's local coordinate system (LCS) is created by specifying the "u" and "w" directions, where the "v" direction is derived from it (we use the right-hand coordinate system, hence v = w x u). The position of LCS's origin is obtained by adding the relating quantity's origin vector (in the case of the beam, it is a column) to the relative position vector (depends on the component)

This arrangement takes care of defining any LCS around the space defined by a global coordinate system. Fig 11 shows the definition of the beam's origin and its local coordinate system.

```
def call_3DModel(self, flag, module_class):  # Done

    self.module_class = module_class

    if self.mainmodule == "Shear Connection":

        A = self.module_class()

        self.loc = A.connectivity

        if flag is True:

            if self.loc == CONN_CWBW:
                self.connectivityObj = self.create3DColWebBeamWeb()

            elif self.loc == CONN_CFBW:
                self.connectivityObj = self.create3DColFlangeBeamWeb()
```

Figure 6: "call_3DModel" method in CommonDesignLogic class

```
class ColFlangeBeamWeb(object):

    def __init__(self, column, beam, angle, nut_bolt_array,gap):
        self.column = column
        self.beam = beam
        self.angle = angle
        self.angleLeft = copy.deepcopy(angle)
        self.nut_bolt_array = nut_bolt_array
        self.gap = gap
```

Figure 7: Portion of __init__of ColFlangeBeamWeb class

```
def create_3dmodel(self):
    self.create_column_geometry()
    self.create_beam_geometry()
    self.create_angle_geometry()
    self.create_nut_bolt_array()

    # Call for create_model
    self.columnModel = self.column.create_model()
    self.beamModel = self.beam.create_model()
    self.angleModel = self.angle.create_model()
    self.angleLeftModel = self.angleLeft.create_model()
    self.nutboltArrayModels = self.nut_bolt_array.create_model()
```

Figure 8: "create_3dmodel" method of ColFlangeBeamWeb class, where all models are separately created. The first four classes are explained in Section 2.4.

```
angle = Angle(L=A.cleat.height, A=A.cleat.leg_a_length, B=A.cleat.leg_b_length, T=A.cleat.thickness,
              R1=A.cleat.root_radius, R2=A.cleat.toe_radius)
```

Figure 9: Angle creation inside connectivity function for Cleat Angle

```
angle = Angle(L, A, B, T, R1, R2)
_place = angle.place(origin, uDir, wDir)
point = angle.computeParams()
prism = angle.create_model()
display.DisplayShape(prism, update=True)
display.DisableAntiAliasing()
start_display()
```

Figure 10: __main__code of angle.py

```
    def create_beam_geometry(self):
        beam_origin = ((self.column.sec_origin + self.column.D / 2) * (-self.column.vDir)) +
(self.column.length / 2 * self.column.wDir) + (self.gap * (-self.column.vDir))
        uDir = numpy.array([0, 1.0, 0])
        wDir = numpy.array([1.0, 0, 0.0])
        self.beam.place(beam_origin, uDir, wDir)
```

Figure 11: Beam geometry creation in main cad file

Chapter 3

Introduction to IFC geometry

In this section, IFC elements are discussed in greater detail. The arguments that must be passed to each IFC element can be obtained either from the documentation or through the getattr.py in the "testing" directory of the "ifcexporter" directory. The output of getattr.py is shown for the IFC element "IfcBeam" in Fig 12. One can see the same "IfcBeam" in [3] and relate to it.

To run getattr.py for an element (Beam), run "python getattr.py IfcBeam" from the command line.

```
(<attribute GlobalId: <type IfcGloballyUniqueId: <string>>>,
 <attribute OwnerHistory: <entity IfcOwnerHistory>>,
 <attribute Name?: <type IfcLabel: <string>>>,
 <attribute Description?: <type IfcText: <string>>>,
 <attribute ObjectType?: <type IfcLabel: <string>>>,
 <attribute ObjectPlacement?: <entity IfcObjectPlacement>>,
 <attribute Representation?: <entity IfcProductRepresentation>>,
 <attribute Tag?: <type IfcIdentifier: <string>>>)
```

Figure 12: Output of getattr.py for "IfcBeam", showing attributes and types or entities

The only main attribute to observe here is the "Representation" and "ObjectPlacement" ("ObjectPlacement" defines the placement) and their respective entity types. The following sub-sections discuss these in detail.

3.1 Representation

For example, the entire representation and placement of the IFC element "IfcBeam" is taken care of by the following code shown below.

Now we will strip this code apart, Note that one can always view the attributes for each element from getattr.py.

#39 defines the ISection Profile of the Beam, where #38 creates the 2D coordinate system as a reference for #39.

#40 extrudes the profile which is then given as an input to #41 which is the shape representation. Note that the type of shape is given as "Swept Solid". The key takeaway here is that there are other complex ways of representing geometry, one such is through 'Boundary representation' or 'Brep'.

#41 is the product definition which is different from shape representation in a way that it can include elements more than the shape such as an axis for the shape. A good example is a cylinder with an axis (product) which is a combination of a cylinder (shape) + axis (a curve). In our case we need no such axis or datum references, hence it takes #40 as the only argument.

The remaining lines are explained in the next section.

```
#38=IFCAXIS2PLACEMENT2D(#9,#6);
#39=IFCISHAPEPROFILEDEF(.AREA.,'IShapedProfile',#38,50.,150.,3.,4.6,None);
#40=IFCEXTRUDEDAREASOLID(#39,#10,#8,500.);
#41=IFCSHAPEREPRESENTATION(#11,'Body','SweptSolid',(#40));
#42=IFCPRODUCTDEFINITIONSHAPE(None,None,(#41));
#43=IFCCARTESIANPOINT((0.,75.,500.));
#44=IFCAXIS2PLACEMENT3D(#43,#7,#6);
#45=IFCLOCALPLACEMENT(#36,#44);
#46=IFCBEAM('0iPvIuMcH9cvkqHFXt1lEJ',#5,'Beam',None,None,#45,#42,None);
```

3.2 Placement

#44 IFCAXIS2PLACEMENT3D must be read as IFC Axis to Placement 3D, it takes in a cartesian point (#43), and new "z" and "x" directions are taken in order, where the "y" direction is derived from it. #45 defines the new "placement" with respect to the parent LCS (#36). We can note that, In the above case, the new "z" direction of #44 is the "y" direction of its parent LCS (#36).

Finally, both placement and representation information is passed on to the "IfcBeam" as we mentioned earlier.

Chapter 4

Design of IFC exporter

The file structure of the IFC exporter is very similar to that of Osdag cad creation. All the IFC handling is done inside the "ifcexporter" directory which will be referred to it as just ifcexporter from now [7]. One can find a similar structure to Osdag inside the "IfcCadModules" directory of ifcexporter. Note that each main cad file has a dedicated file inside ifcexporter for its IFC export, this file will be referred to as the main IFC file.

Similar to "items" in cad, we have "IfcItems" in ifcexporter. It also contains a parent file containing the parent class for all classes in IFC items, the IfcInitializer.py. It has a parent class "IfcObject" which serves as a parent for the remaining IFC Items. The methods of "IfcObject" will be covered wherever necessary.

There are two known methods of IFC export. The Brep Export method makes use of IfcOpenShell's tesselate feature to convert the PythonOCC TopoDS object into the IfcShapeRepresentation of the "Brep" type. This method is simple to program but comes with the disadvantage of large file size (Note: some viewers can't open the geometry created by this method, say, OpenIFCviewer and FreeCAD).

On the other hand, the SweptSolid Export method produces compact IFC files but it is harder to program, all viewers can view the files produced by this method.

4.1 Introduction to IfcOpenShell

IfcOpenShell is an open-source IFC toolkit and geometry engine, available for both python and c++. It is installed in the conda virtual environment (see site-packages), one can install it from here [8]

IfcOpenShell provides an easy way to write to an IFC file and also contains some very useful functions like "tesselate" which is used in the Brep Export method [9]. To make it further easier, the "IfcObject" class is written which any class can inherit its properties from.

4.2 IfcInitializer.py

This file is one of the most important files for creating IFC files as it contains the class "IfcObject" which is the parent class for the following classes. The main functions of this class are discussed in this section.

In the __init__ of this class we have the logic for whether to create an ifcfile object (Not to confuse it with an actual file. It's an IfcOpenShell object) or use the previously created ifcfile.

```
class IfcObject():
    def __init__(self, ifcfile = None, **kwgs):
        if ifcfile == None:
            self.create_ifcfile(**kwgs)
        else:
            self.ifcfile = ifcfile
            self.extract_data_from_ifctemplate()
```

Figure 13: __init__ of IfcObject class

In the create_ifcfile() method, we create the template for the IFC file using the given keyword arguments and move on to build the initial hierarchy. The hierarchy must look like this, The code for it is pretty much self-explanatory.

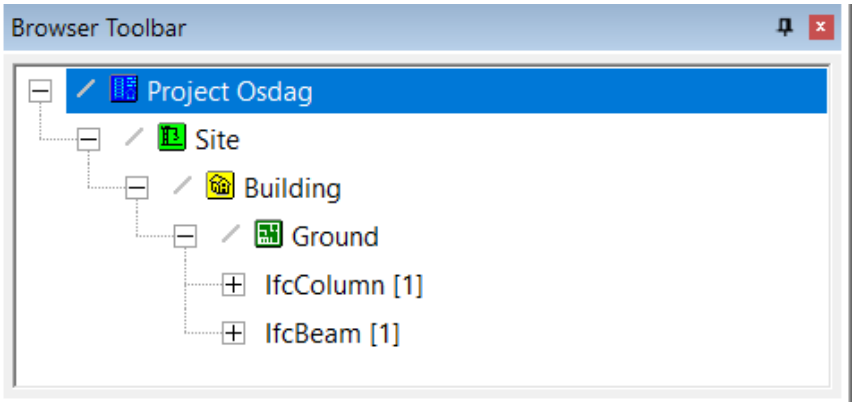

Figure 14: Hierarchy of IFC creation of Osdag

https://www.overleaf.com/project/6302492a735371c393634490
This class contains other important methods too, which are explained on the fly.

4.3 ui_template.py

The specific IFC module to run when a user clicks export is covered in the following conditional, this can be found here [10]

```
elif file_extension == 'ifc':
    ifc_class = self.get_ifc_class(main.module, main.mainmodule)
    CONFIG = {
        "filename": os.path.basename(os.path.normpath(fName)),
    }
    ifcobj = ifc_class(**CONFIG)
    ifcobj.ifc_write_to_path(fName)
```

Figure 15: Code for saving the IFC file

```
def get_ifc_class(self, module, mainmodule):
    module_class = self.return_class(module)
    A = module_class()
    loc = A.connectivity
    if mainmodule == "Shear Connection":
        if loc == CONN_CFBW:
            return CleatAngle_colFlangeBeamWebConnectivity
        if loc == CONN_CWBW:
            return CleatAngle_colWebBeamWebConnectivity
```

Figure 16: Code for getting the appropriate IFC class for export

Chapter 5

Brep Export Method

In this method first, The models created in the cad main file are written as .brep files in the "Temp" directory of ifcexporter. The following addition must be made to the create_3dmodel() method of the main cad file, see the main cad file here [11]

```
from ifcexporter.IfcItems.IfcInitializer import temp_brep
```

Figure 17: Import of temp_brep function from ifcexporter in main cad file

```
ifc_class_name = "CleatAngle_colFlangeBeamWebConnectivity"

self.aux_data = {} # For Ifc_export

temp_brep(self.columnModel, "column")
temp_brep(self.beamModel, "beam")
temp_brep(self.angleModel, "angle")
temp_brep(self.angleLeftModel, "angleLeft")
for i, model in enumerate(self.nut_bolt_array.models):
    temp_brep(model, "fastener" + str(i))

self.aux_data["no_of_fasteners"] = len(self.nut_bolt_array.models)

json.dump(self.aux_data, open("ifcexporter/Temp/"+ ifc_class_name + ".json",'w'))
```

Figure 18: Addition for IFC export to create_3dmodel() of main cad file

Here we import the temp_brep from IfcInitializer.py. This function writes a TopoDS class as brep file with the given name. (see Fig 17, 19)
Now, in the IFC main file, these brep files are converted into IFC objects using the IFC items, in the following way,
In the main IFC file, the necessary IFC items for geometry creation are imported (see Fig 20)

```
def temp_brep(prism, name):
    fname = "ifcexporter/Temp/" + name + ".brep"
    breptools.Write(prism, fname)
```

Figure 19: temp_brep function which uses breptools of PythonOCC

```
from ifcexporter.IfcItems.IfcInitializer import *
from ifcexporter.IfcItems.ISection import ISection
from ifcexporter.IfcItems.Plate import Plate
from ifcexporter.IfcItems.Fastener import Fastener
```

Figure 20: Imports in the main IFC file

Then in the __init__ method, __init__ of IfcObjectis first called (see Fig 21), which initializes the ifcfile object. Then the auxiliary data stored in the "Temp" directory is read and the create_models() method is called.

```
class CleatAngle_colFlangeBeamWebConnectivity(IfcObject):
    def __init__(self, **kwgs):
        super().__init__(**kwgs)
        ifc_class_name = self.__class__.__name__
        self.aux_data = json.load(open("ifcexporter/Temp/" + ifc_class_name + ".json"))
        self.create_models()
        self.clear_Temp()
```

Figure 21: __init__ of main IFC file/class

The create_models() method calls the geometry creation of each element that makes use of the IFC items. Section 5.1 covers IFC items.

5.1 IFC items

Currently, Osdag's default method of IFC export is the Brep export method. Hence all the items in IfcItems are written with respect to the Brep export method. The "RawDesign" directory contains items for SweptSolid export method but is not fully developed.

Here, IFC items will be explained with ISection example,

We saw in Section 3 that representation is necessary to form an IFC element, here we use the get_rep() method which takes in the name of the element, and reads the .brep file with that name from "Temp". (see Fig 24)

The get_rep() method uses ifcopenshell's "geom" module (here imported as ifcgeom), for converting the TopoDS class into IFC representation.

```
def create_models(self):
    self.create_column()
    self.create_beam()
    self.create_angle()
    self.create_angleLeft()
    self.create_fasteners()

def create_column(self):
    ISection(self.ifcfile, "column", type = "column")

def create_beam(self):
    ISection(self.ifcfile, "beam", type = "beam")

def create_angle(self):
    Plate(self.ifcfile, "angle")

def create_angleLeft(self):
    Plate(self.ifcfile, "angleLeft")
```

Figure 22: create_models() method of main IFC file/class

```
class ISection(IfcObject):
    def __init__(self, ifcfile, name, placement = None, storey = None, type = "column", **kwgs):
        super().__init__(ifcfile)
        self.name = name
        self.product_rep = self.get_rep(name)
        if placement == None:
            self.placement = self.ifcfile.ground_storey_placement
        self.type = type
        self.process_kwgs(kwgs)
        ifcobj = self.create_ifcobj()
        self.assign_storey(ifcobj, storey)
```

Figure 23: ISection class in IfcItems

```
def read_brep(self, fname):
    shape = TopoDS_Shape()
    builder = BRep_Builder()
    breptools.Read(shape, fname, builder)
    return shape

def get_rep(self, name):
    fname = "ifcexporter/Temp/" + name + ".brep"
    shape = self.read_brep(fname)
    representation = ifcgeom.tesselate(self.ifcfile.schema, shape, 1.)
    self.ifcfile.add(representation)
    return representation
```

Figure 24: get_rep method of IfcObject.

Chapter 6

SweptSolid Export Method

The files inside the "RawDesign" directory of IFC items will be used in this method. Here, instead of using TopoDS classes, Location and Design data are directly obtained from the main cad file and they are used to produce the geometry from scratch using geometry creation methods in IfcInitializer such as create_ifcextrudedareasolid() for profile extrusion and create_arbitraryclosedprofiledef() for profile creation.

Although this method produces compact files, the amount of work that needs to be done per module or connectivity is high compared to the Brep Export method. The following sections give an overview of how these data are collected.

6.1 Extraction of Location data

While each geometry gets created in the main cad file, an attribute called "location" is assigned to the geometry object itself.

```
def create_column_geometry(self):

    column_origin = numpy.array([0, 0, 0])
    column_u_dir = numpy.array([0, 1.0, 0])
    wDir1 = numpy.array([0.0, 0, 1.0])
    self.column.location = numpy.array([column_origin, column_u_dir, wDir1])
    self.column.place(column_origin, column_u_dir, wDir1)
```

Figure 25: Assignment of location attribute to column object

Now, this can be read back in create_3dmodel() and is written to a .json file.

```
def create_3dmodel(self):
    self.create_column_geometry()
    self.create_beam_geometry()
    self.create_angle_geometry()
    self.create_nut_bolt_array()

    location_data = dict(
        column = self.column.location.tolist(),
        beam = self.beam.location.tolist(),
        angle = self.angle.location.tolist(),
        angleLeft = self.angleLeft.location.tolist()
    )

    json.dump(location_data, open("ifcexporter/LocationData/CleatColFlangeBeamWebConnectivity.json",'w'))
```

Figure 26: Reading and writing of location data of all cad items.

6.2 Extraction of Design data

Instead of extracting from main cad file, the design data can be directly obtained from common_logic.py's connectivity function (see Fig 27). Then finally, all the design data is packed in a single dictionary as shown in Fig 28

```
angle_data = dict(
    L=A.cleat.height,
    A=A.cleat.leg_a_length,
    B=A.cleat.leg_b_length,
    T=A.cleat.thickness,
    R1=A.cleat.root_radius,
    R2=A.cleat.toe_radius
)
```

Figure 27: Packing the design parameters of the "Angle" cad item in a single dictionary.

The type of design data one collects may vary, for instance one can collect all the profile points required to create the angle profile directly. Fig 29 and 30 show how.

The tolist() method is necessary here because numpy arrays cannot be written to a .json file.

6.3 Representation

The Product representation is created in the Angle class __init__ itself. The profile is created first followed by extrusion. The method "create_angle_as_ifcplate" requires the name and placement of the angle which is provided in the main IFC file (see Fig 31)

```
ifc_export_data = {
    "angle": angle_data,
    "gap": gap_data,
    "bolt": bolt_data,
    "nut": nut_data,
    "supported": supported_data,
    "supporting": supporting_data,
    "nut_space": nut_space_data,
    "cnut_space": cnut_space_data
}

json.dump(ifc_export_data, open("ifcexporter/DesignData/CleatColFlangeBeamWebConnectivity.json",'w'))
```

Figure 28: Design data/dictionaries of all the cad items in a single dictionary ifc_export_data

```
angle = Angle(**angle_data)
```

Figure 29: The angle cad item is created from angle data

The create_angle_as_ifcplate() method requires one to see the attributes taken by the IfcPlate element and write the "params" dictionary manually. (see Fig 32)

6.4 Placement

The placement is done by the place() method of the IfcObject. It takes four arguments, parent coordinate system, new origin, new "z" and new "x" direction. The parent coordinate system is generally the storey's coordinate system.

One can relate to the same function in the main IFC file used in the Brep export method to see the differences.

```
angle_data.update({"profile_coords": [i.tolist() for i in angle.points]})
```

Figure 30: After angle cad item creation, angle.points that contain profile coordinates as numpy array is appended to the angle_data dictionary as lists.

```
class Angle(IfcObject):
    def __init__(self, ifcfile, **kwgs):
        super().__init__(ifcfile)
        self.process_kwgs(kwgs)
        self.construct_profile()
        self.extrude_body()
        self.product_rep = self.create_product_representation(self.extruded_body)

    def process_kwgs(self, kwgs):
        self.height = kwgs.get("L")
        pc = kwgs.get("profile_coords")
        pc.append(pc[0])
        self.profile_coords = pc

    def construct_profile(self):
        self.profile_placement = self.ifcfile.createIfcAxis2Placement2D(self.ori, self.Xdir)
        self.profile = self.create_ifcarbitraryclosedprofiledef(self.profile_coords)

    def extrude_body(self):
        self.extruded_body = self.ifcfile.createIfcExtrudedAreaSolid(self.profile, self.g_placement, self.Zdir, self.height)

    def create_angle_as_ifcplate(self, name, placement):
        params = {
            "GlobalId": self.guid(),
            "Name": name,
            "OwnerHistory": self.owner_history,
            "ObjectPlacement": placement,
            "Representation": self.product_rep
        }
        ifccolumn = self.ifcfile.createIfcPlate(**params)
        return ifccolumn
```

Figure 31: The typical IFC item file for the SweptSolid export method.

```
(<attribute GlobalId: <type IfcGloballyUniqueId: <string>>>,
 <attribute OwnerHistory: <entity IfcOwnerHistory>>,
 <attribute Name?: <type IfcLabel: <string>>>,
 <attribute Description?: <type IfcText: <string>>>,
 <attribute ObjectType?: <type IfcLabel: <string>>>,
 <attribute ObjectPlacement?: <entity IfcObjectPlacement>>,
 <attribute Representation?: <entity IfcProductRepresentation>>,
 <attribute Tag?: <type IfcIdentifier: <string>>>)
```

Figure 32: Attributes of IFC plate

```
def create_angle(self):
    self.angle = Angle(self.ifcfile, **self.angle_dd)
    self.angle_placement = self.place(location = self.angle_ld)
    self.ifcplate_angle = self.angle.create_angle_as_ifcplate("Angle", self.angle_placement)
```

Figure 33: Placement is done in the main IFC file create_angle() method

Chapter 7

Ifc Property Sets

IFC property sets are properties that are added to IFC elements. For example, A column may have its profile data stored in a property set called "Pset_ProfileData" (see Fig 34)

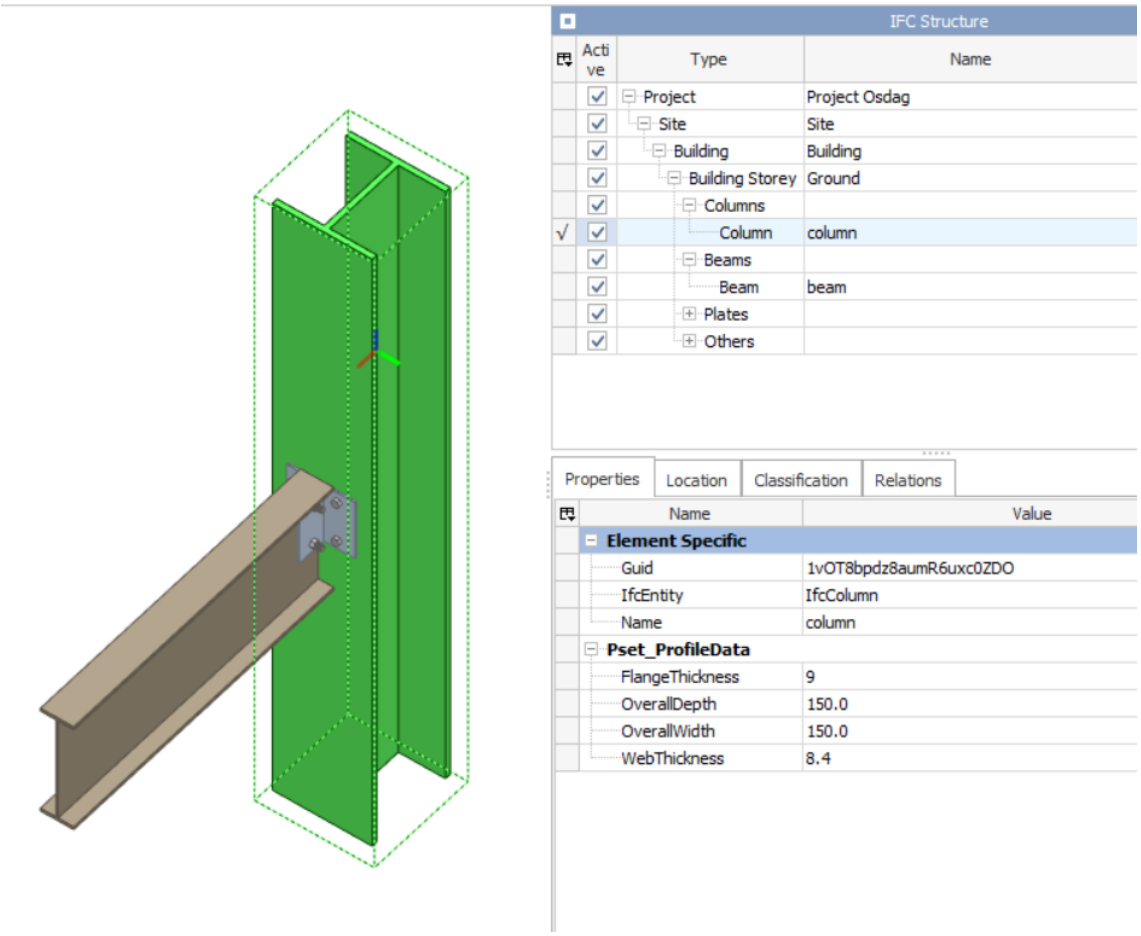

Figure 34: Pset_ProfileData of an IFC column.

The following lines of the IFC file can create the above-seen property set.

#238=IFCCOLUMN('1vOT8bpdz8aumR6uxc0ZDO',#5,'column',None,None,#25,#237,None);
#239=IFCPROPERTYSINGLEVALUE('OverallWidth',None,IFCTEXT('150.0'),None);
#240=IFCPROPERTYSINGLE-
VALUE('OverallDepth',None,IFCTEXT('150.0'),None);
#241=IFCPROPERTYSINGLE-
VALUE('WebThickness',None,IFCTEXT('8.4'),None);
#242=IFCPROPERTYSINGLE-
VALUE('FlangeThickness',None,IFCTEXT('9'),None);
#243=IFCPROPERTY-
SET('0vplNYSpTF3gISF4_GN2V',#5,'Pset_ProfileData',None,(#239,#240,#241,#242);
#244=IFCRELDEFINESBYPROPER-
TIES('3kJJypZI50eQcCQYfFvxC1',#5,None,None,(#238),#243);

The first four lines from #239 define the properties themselves and their values as IfcText, although IFC provides a lot of flexibility here to save it in any format as we wish. #243 packs them into a single property set called "Pset_ProfileData". It's a convention to add "Pset_" as a suffix. #244 links the property set to the column.

The above can be achieved via code through the assign_Pset() method of IfcObject (see Fig 35). The profile data can be obtained from the main cad file [12].

```
def assign_Pset(self, ifcobj, Pset_name, props):
    ifc_prop_single_values = []
    for k, v in props.items():
        v = str(v)
        psv = self.ifcfile.createIfcPropertySingleValue(k, None, self.ifcfile.createIfcText(v), None)
        ifc_prop_single_values.append(psv)
    Pset = self.ifcfile.createIfcPropertySet(
        self.guid(),
        self.owner_history,
        Pset_name,
        None,
        ifc_prop_single_values
    )
    self.ifcfile.createIfcRelDefinesByProperties(
        self.guid(),
        self.owner_history,
        None,
        None,
        [ifcobj],
        Pset
    )
```

Figure 35: assign_Pset() method of IfcObject

This method is then called in the respective IFC item for which property sets are necessary.

Chapter 8

Appendix

8.1 List of IFC viewers

1. BIMvision [13]
2. FZK viewer [14]
3. Open IFC viewer [15]
4. IfcQuery - Open Source [16]
5. Trimble Connect (previously Tekla BIMsight) [17]

References

1. IFC - wikipedia (https://en.wikipedia.org/wiki/Industry_Foundation_Classes)
2. buildingSMART (https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes)
3. Sample IFC file (https://github.com/MukunthAG/Osdag/blob/master/ifcexporter/Testing/Samples/CleatColFlangeBeamWebConnectivity_stage1.ifc)
4. Revit's IFC Documentation (https://damassets.autodesk.net/content/dam/autodesk/draftr/2528/180213_IFC_Handbuch.pdf)
5. buildingSMART IFC Documentation (https://standards.buildingsmart.org/IFC/RELEASE/IFC2x3/TC1/HTML/)
6. create3DColFlangeBeamWeb method of CommonDesignLogic class (https://github.com/osdag-admin/Osdag/blob/master/cad/common_logic.py#L554)
7. IFC exporter updated repository (https://github.com/MukunthAG/Osdag/tree/master/ifcexporter)
8. IfcOpenShell (http://ifcopenshell.org/python) IfcOpenShell code examples (https://wiki.osarch.org/index.php?title=IfcOpenShell_code_examples)
9. ui_template.py IFC export conditional (https://github.com/MukunthAG/Osdag/blob/master/gui/ui_template.py#: :text=elif
10. Main cad file with IFC export function (https://github.com/MukunthAG/Osdag/blob/master/cad/ShearConnections/CleatAngle/colFlangeBeamWebConnectivity.py.)
11. Property set export from main cad file (https://github.com/MukunthAG/Osdag/blob/master/cad/ShearConnections/CleatAngle/colFlangeBeamWebConnectivity.py#L80)
12. BIMvision (https://bimvision.eu/)
13. FZK viewer (https://www.iai.kit.edu/english/1302.php)
14. Open IFC viewer (https://openifcviewer.com/)
15. IfcQuery (https://ifcquery.com/#: :text=house
16. Trimble connect (https://www.tekla.com/products/tekla-bimsight)