CIMantic Graphs

Release v1_0_0a

Alex Anderson & CIMantic Graphs Team

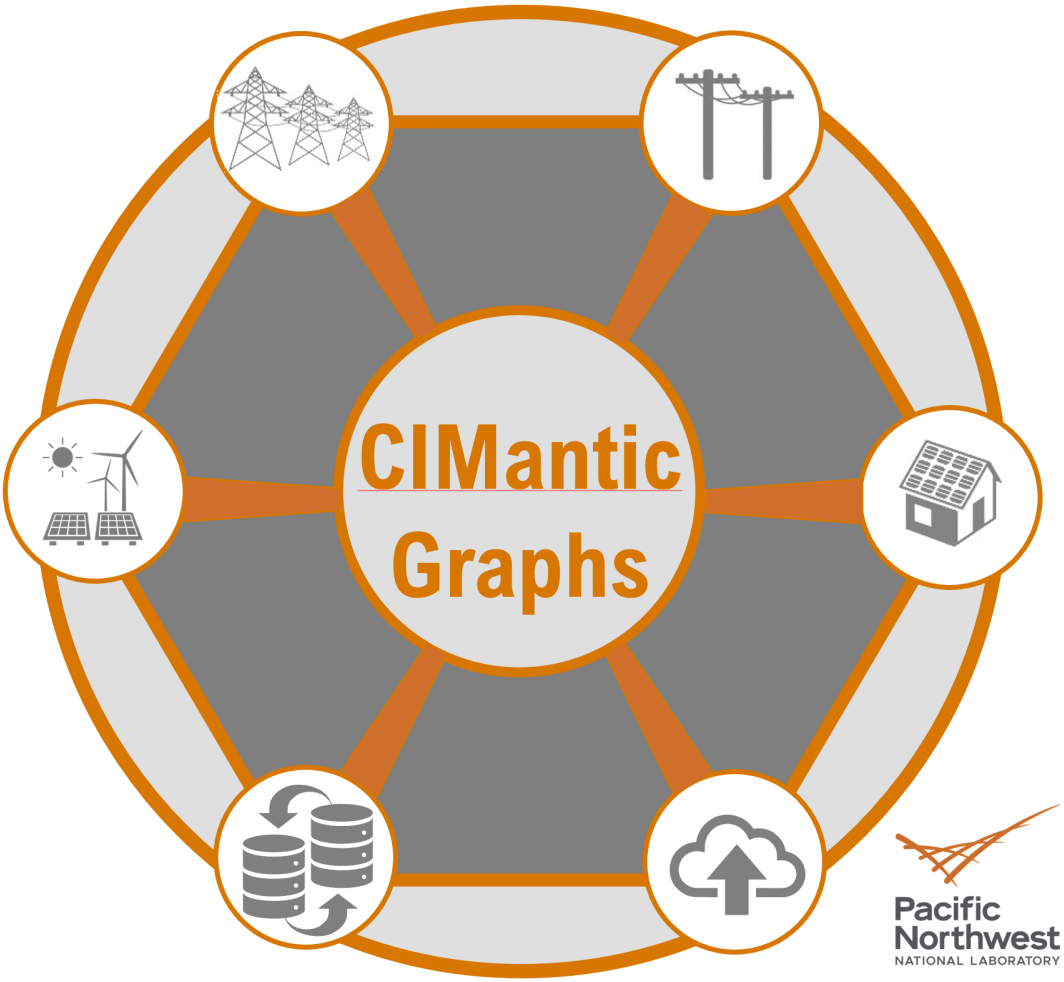
Oct 13, 2023

OVERVIEW

Python library for creating in-memory labeled property graphs for creating, parsing, and editing CIM power system models. It creates Python object instances in memory using a data profile exported from a specified CIM profile (e.g. IEC61970cim18v01 or GridAPPS-D CIM100 RC4_2021).

The library is being expanded to cover centralized applications, transmission models, and real-time editing of CIM XML models natively.

To install CIMantic Graphs clone the github repository or use pip install: pip install cim-graph



CHAPTER

ONE

CIMANTIC GRAPHS OVERVIEW

This tutorial provides an introduction to usage of the CIMantic Graphs library (aka CIM-Graph).

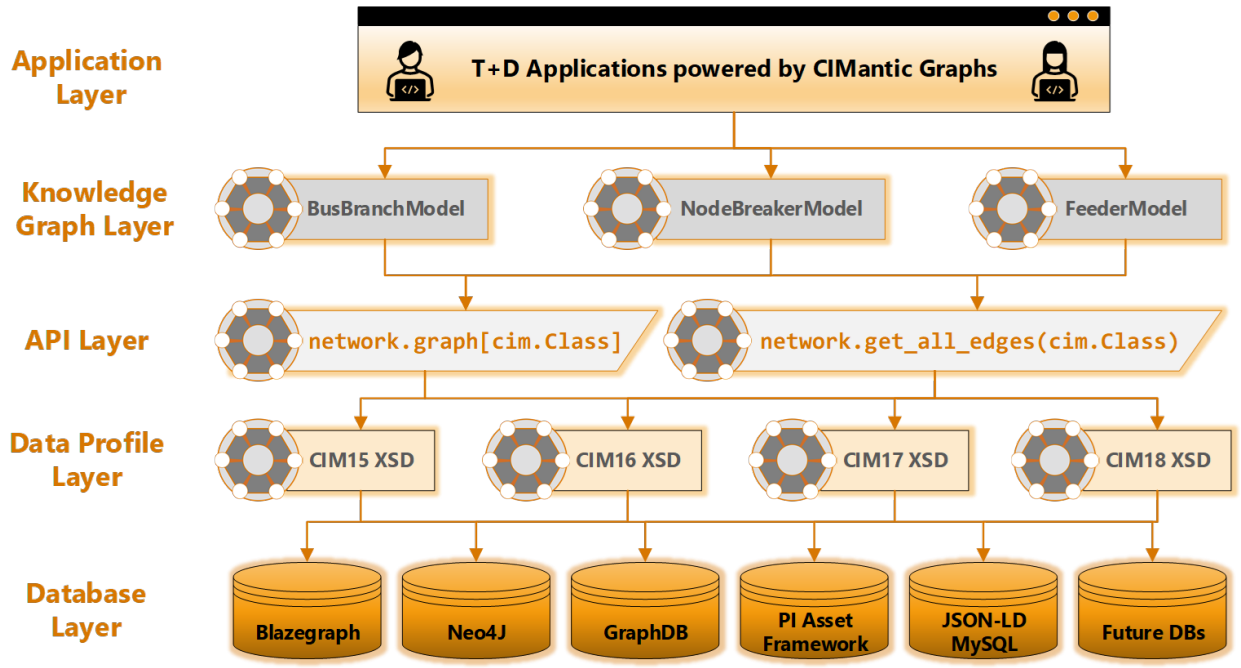
CIMantic Graphs is an open-source library for creating in-memory labeled property graphs for creating, parsing, and editing CIM power system models. It creates Python object instances in memory using a data profile exported from a specified CIM profile (e.g. IEC61970cim17v40 or GridAPPS-D RC4_2021).

To install CIMantic Graphs clone the github repository or use pip install: `pip install cim-graph`

1.1 Table of Contents:

1.2 1. CIMantic Graphs Structure

CIMantic Graphs uses the layered architecture shown below:



1.2.1 1.1. Database Layer

CIMantic Graphs currently supports the following databases and interfaces:

- Blazegraph Database
- GraphDB Database
- Neo4J Database
- MySQL Database (in progress)
- GridAPPS-D Platform
- AVEVA PI Asset Framework (in progress)
- RDFLib File Parser

The library uses a unified syntax for all upper-level calls and routines. The databases can be swapped interchangeably by changing the `ConnectionParameters` created during application startup and no other changes to any other application syntax or methods

1.2.2 1.2. Data Profile Layer

CIMantic Graphs is able to support any standard or custom CIM profile. The CIM profile needs to be exported as an XSD data profile / schema. CIMantic Graphs is then able to ingest the data profile and convert all UML classes and attributes to python dataclasses, which power all of the routines and unified API syntax

1.2.3 1.3. API Layer

CIMantic Graphs offers a breakthrough in terms of ease-of-use through a unified API with two core methods.

Access to labeled property graph objects:

- `network.graph[cim.ClassName]`: This offers access to a catalog of CIM object instances stored in memory and sorted by class type and mRID forming the named property graph.

Universal database query method:

- `network.get_all_edges(cim.ClassName)`: This is a universal query method that gets all attributes and all objects one edge away from instances of the specified class. This method works for all CIM classes, CIM profiles, serialization formats, and supported databases.

1.2.4 1.4. Knowledge Graph Layer

CIMantic Graphs offers three core knowledge graph classes for handing various kinds of power system models:

- `BusBranchModel`: Transmission bus-branch models commonly used for planning and power flow studies
- `NodeBreakerModel`: Transmission node-breaker models commonly used inside energy management systems
- `FeederModel`: Distribution feeder models with support for single-phase unbalanced networks used in North America.

Centralized or distributed representations of the power system network model can be used. Centralized models use a single labeled property graph for the network. Distributed models use nested `DistributedArea` class instances to represent the grouping of equipment inside a Substation, VoltageLevel, Bay, Feeder, and switch-delimited topological area inside a combined T+D model.

1.2.5 1.5. Application Layer

T+D applications are able to access all of the power system objects through knowledge graph, without any need to connect to the database or perform any custom i/o operations.

1.3 2. Importing CIMantic Graphs

CIMantic Graphs contains several modules within the core library:

- `data_profiles`: This package contains full CIM profiles exported through CIMTool or Enterprise Architect Schema Composer.
- `databases`: This package contains database i/o connections and backend query handling for multiple databases (e.g. Blazegraph, GraphDB, Neo4J, MySQL, etc.)
- `models`: This package contains knowledge graph models for transmission node-breaker, transmission bus-branch, and distribution feeder models.
- `queries`: This package contains generic queries that are built at runtime for any CIM profile and CIM class

1.3.1 2.1. Importing the CIM Profile

The first step in using CIMantic Graphs is to import the python data profile for desired CIM profile. The data profile provides the ability to invoke CIM classes directly based on their name.

Method 1: Directly import the desired CIM profile:

[1]:
```
import cimgraph.data_profile.rc4_2021 as cim
```

Method 2 (recommended): Use `importlib`:

[2]:
```
import importlib
cim_profile = 'rc4_2021'
cim = importlib.import_module('cimgraph.data_profile.' + cim_profile)
```

1.3.2 2.2. Invoking CIM classes

After importing the data profile, it is possible to create an instance of a class or view the attributes of any class.

Example 1: Create a new breaker with a name and mRID. The `uuid` library can be used to generate the unique identifier used for all CIM objects.

[3]:
```
import uuid
breaker = cim.Breaker(name = "breaker_01", mRID = str(uuid.uuid4()))
breaker.open = True
print(breaker)
```

```
Breaker(mRID='597b1182-bfdc-4561-9a3c-7138c4259bde', aliasName=None, description=None,
↪ name='breaker_01', Names=[], AssetDatasheet=None, Assets=[], ConfigurationEvent=[],
↪ Controls=[], Location=None, Measurements=[], OperatingShare=[], PSRType=None,␣
↪ReportingGroup=[], aggregate=None, inService=None, networkAnalysisEnabled=None,␣
↪normallyInService=None, AdditionalEquipmentContainer=[], EquipmentContainer=None,␣
↪Faults=[], OperationalLimitSet=[], UsagePoints=None, BaseVoltage=None, SvStatus=[],␣
↪Terminals=[], normalOpen=None, open=True, ratedCurrent=None, retained=None,␣
↪CompositeSwitch=None, SvSwitch=[], SwitchPhase=[], SwitchSchedules=[],␣
↪breakingCapacity=None, inTransitTime=None)
```

Example 2: Associate two CIM objects based on their associations

[4]:
```
substation = cim.Substation(name = "sub_1", mRID = str(uuid.uuid4()))
breaker.EquipmentContainer = substation
```

Example 3: View documentation of the ACLineSegment class

[5]:
```
print(cim.ACLineSegment.__doc__)
```

```
A wire or combination of wires, with consistent electrical
    characteristics, building a single electrical system, used to carry
    alternating current between points in the power system.

    For symmetrical, transposed 3ph lines, it is sufficient to use
    attributes of the line segment, which describe impedances and
    admittances for the entire length of the segment.  Additionally
    impedances can be computed by using length and associated per length
    impedances. The BaseVoltage at the two ends of ACLineSegments in a
    Line shall have the same BaseVoltage.nominalVoltage. However,
    boundary lines  may have slightly different
```

(continues on next page)

(continued from previous page)

```
    BaseVoltage.nominalVoltages and  variation is allowed. Larger
    voltage difference in general requires use of an equivalent branch.

    :ivar b0ch: Zero sequence shunt (charging) susceptance, uniformly
        distributed, of the entire line section.
    :ivar bch: Positive sequence shunt (charging) susceptance, uniformly
        distributed, of the entire line section.  This value represents
        the full charging over the full length of the line.
    :ivar g0ch: Zero sequence shunt (charging) conductance, uniformly
        distributed, of the entire line section.
    :ivar gch: Positive sequence shunt (charging) conductance, uniformly
        distributed, of the entire line section.
    :ivar r: Positive sequence series resistance of the entire line
        section.
    :ivar r0: Zero sequence series resistance of the entire line
        section.
    :ivar shortCircuitEndTemperature: Maximum permitted temperature at
        the end of SC for the calculation of minimum short-circuit
        currents. Used for short circuit data exchange according to IEC
        60909
    :ivar x: Positive sequence series reactance of the entire line
        section.
    :ivar x0: Zero sequence series reactance of the entire line section.
    :ivar ACLineSegmentPhases: The line segment phases which belong to
        the line segment.
    :ivar Clamp: The clamps connected to the line segment.
    :ivar Cut: Cuts applied to the line segment.
    :ivar LineFaults: The line faults of the line segment.
    :ivar ParallelLineSegment:
    :ivar PerLengthImpedance: Per-length impedance of this line segment.
    :ivar WireSpacingInfo:
```

1.4 3. Specifying Connection Parameters

The next step in using any of the CIMantic Graphs library classes to establish the connection parameters for reading or writing the CIM model. The `ConnectionParameters` class can be imported through:

[6]:
```
from cimgraph import ConnectionParameters
```

The parameters can be defined using an instance specifying the required paramters for the specific connection interface (database) to be used.

[7]:
```
params = ConnectionParameters(url = "http://localhost:8889/bigdata/namespace/kb/sparql
↪",
                              cim_profile='rc4_2021', iec61970_301=8) # Blazegraph␣
↪params
```

The required and optional arguments for the `ConnectionParameters` class are described in detail in Connection-Parameters Arguments

1.5 4. Connecting to the Data Source

The next step is to connect to the database or file source that will be used for CIM model. A complete list of connection types currently supported are described in Supported Databases

[8]:
```
from cimgraph.databases.blazegraph import BlazegraphConnection
blazegraph = BlazegraphConnection(params)
```

1.6 5. Creating a Container and Graph Model

CIMantic Graphs uses an EquipmentContainer class as the starting point for building a knowledge graph of the power system model. The supported classes are `BusBranchModel`, `NodeBreakerModel`, and `FeederModel`.

[9]:
```
from cimgraph.models import FeederModel
```

The power system network is then created by passing the database connection, container object, and a boolean flag whether a centralized or distributed model should be built.

[10]:
```
feeder_mrid = "49AD8E07-3BF9-A4E2-CB8F-C3722F837B62"  # 13 bus
feeder = cim.Feeder(mRID=feeder_mrid)
network = FeederModel(connection=blazegraph, container=feeder, distributed=False)
```

The network is populated by default with all ConductingEquipment, ConnectivityNode, and Terminal class instances. The knowledge graph is indexed by the class type and then the device mRID.

To view instances of a particular class, use the `.pprint(cim.ClassName)` method. Use of the default python print method is not recommended due to foward-reverse relationships resulting in an infinite print loop.

[11]:
```
network.pprint(cim.Breaker)
```

```
{
    "52DE9189-20DC-4C73-BDEE-E960FE1F9493": {
        "mRID": "52DE9189-20DC-4C73-BDEE-E960FE1F9493",
        "Terminals": [
            "1D81C7FE-E88F-41E3-A900-476CA6476CCD",
            "2847E06B-C8ED-41E6-B515-C61C9E8EB4B4"
        ]
    }
}
```

[ ]:
```

```

1.7 6. Automated Database Queries

CIMantic Graphs contains automatic query generation routines for all classes and all supported databases using the `.get_all_edges(cim.ClassName)` method. This query obtains all attributes for all objects of that class type and expands the knowledge graph by one edge with default instances of all associated objects.

[12]:
```
network.get_all_edges(cim.Breaker)
network.get_all_edges(cim.Terminal)
network.get_all_edges(cim.ConnectivityNode)
```

[13]:
```
network.pprint(cim.Breaker)
```

```
{
    "52DE9189-20DC-4C73-BDEE-E960FE1F9493": {
        "mRID": "52DE9189-20DC-4C73-BDEE-E960FE1F9493",
        "name": "brkr1",
        "Location": "085BBE1F-FF95-4260-A9D2-8D2F1A8EA9A3",
        "Measurements": [
            "8e7f04ee-a032-4128-838e-a3442a1c3026",
            "9f5cb9c4-71d6-4f2b-9dc1-26c7e9f84410",
            "ab18a8e1-f023-4f9e-bf02-c75bf05164df",
            "aec42b89-f3c0-47e9-b21a-82736b2a1149",
            "b393e719-0981-4498-9d96-07f1be7de009",
            "baccfd49-ec98-4380-8be9-d242dc8611f3",
            "f11a9ad9-5b68-483b-b52f-dd4af07bb77d",
            "0c48c74a-ceee-4c99-bd73-28079561ca7a",
            "3c60208a-8ef8-483c-828b-30ee42d402dc",
            "40ac2899-1d2a-469f-a14a-1e14ea29d172",
            "43f80e34-281e-4baa-8aba-d931a9a3ab87",
            "7c6f94c1-1419-4582-ab2d-a0b11772c26b"
        ],
        "EquipmentContainer": "49AD8E07-3BF9-A4E2-CB8F-C3722F837B62",
        "BaseVoltage": "2A158E0C-CD01-4A50-AEBA-59D761FCF15D",
        "Terminals": [
            "1D81C7FE-E88F-41E3-A900-476CA6476CCD",
            "2847E06B-C8ED-41E6-B515-C61C9E8EB4B4"
        ],
        "normalOpen": "false",
        "open": "false",
        "ratedCurrent": "400",
        "retained": "true",
        "breakingCapacity": "400"
    }
}
```

1.8 7. Traversing the Knowledge Graph

CIMantic Graphs associates CIM classes by creating direct references between in-memory object instances based on the naming and hierarchy of the underlying information model.

To view the attributes of particular object instance, directly invoke the attribute from the UML class diagram.

[14]:
```
breaker = network.graph[cim.Breaker]["52DE9189-20DC-4C73-BDEE-E960FE1F9493"]
print(breaker.normalOpen)
```

```
false
```

To traverse the knowledge graph, no custom queries are required. Instead, directly invoke the UML association names that serves as references between objects in the graph:

[15]:
```
print(breaker.Terminals[0].ConnectivityNode.name)
```

```
650
```

No separate queries or mapping are required for measurment objects. Call the `.get_all_edges` method for each measurement class, and then obtain the measurements from the equipment object

[16]:
```
network.get_all_edges(cim.Analog)
network.get_all_edges(cim.Discrete)
```

[17]:
```
for meas in breaker.Measurements:
    p = meas.phases
    print("name:", meas.name, "  phase:" , meas.phases, "  bus:", meas.Terminal.
↪ConnectivityNode.name)
```

```
name: Breaker_brkr1_Voltage   phase: PhaseCode.B   bus: 650
name: Breaker_brkr1_State   phase: PhaseCode.A   bus: 650
name: Breaker_brkr1_Voltage   phase: PhaseCode.C   bus: 650
name: Breaker_brkr1_State   phase: PhaseCode.C   bus: 650
name: Breaker_brkr1_Voltage   phase: PhaseCode.C   bus: brkr
name: Breaker_brkr1_State   phase: PhaseCode.B   bus: 650
name: Breaker_brkr1_Voltage   phase: PhaseCode.A   bus: brkr
name: Breaker_brkr1_Current   phase: PhaseCode.A   bus: 650
name: Breaker_brkr1_Current   phase: PhaseCode.B   bus: 650
name: Breaker_brkr1_Voltage   phase: PhaseCode.A   bus: 650
name: Breaker_brkr1_Current   phase: PhaseCode.C   bus: 650
name: Breaker_brkr1_Voltage   phase: PhaseCode.B   bus: brkr
```

CHAPTER

TWO

CONNECTION PARAMETERS

The first step in using any of CIMantic Graphs functionalities is to define the connection parameters, which specify the CIM Profile, serialization format, and database to be used. The `ConnectionParameters` class is used to specify these inputs with the following required and optional arguments:

[2]:
```
from cimgraph import ConnectionParameters
```

2.1 1. Connection Arguments

Required arguments:

- `cim_profile`: This specifies the specific version of CIM to be used, based on the available python data profiles loaded into the library

Optional arguments:

- `namespace`: CIM namespace, default is `"http://iec.ch/TC57/CIM100#"`
- `iec61970-301`: Serialization version, default is 7
- `url`: URL at which the database can be reached via TCP/IP or other connection
- `host`: Database host address
- `port`: Database host port
- `database`: Database name
- `username`: Database username
- `password`: Database password
- `filename`: Filename for importing CIM models from an XML file

Note that not all parameters are required. Each database connection uses a subset of these arguments depending on the requirements of the database connection driver.

2.1.1 1.1. CIM Namespaces

Each version of the CIM uses a specific namespace, which can typically be found in the first line of the XML file, such as

`xmlns:cim="http://iec.ch/TC57/2011/CIM-schema-cim15#"`

The default used by CIMantic Graphs is the CIM100 namespace `"http://iec.ch/TC57/CIM100#"`

To change the namespace used, pass the namespace as a string into the connection parameters. This will then be used in all automated query builders:

[3]:
```
params = ConnectionParameters(namespace = "http://iec.ch/TC57/2011/CIM-schema-cim15#")
```

2.1.2 1.2. IEC 61970-301 Serialization Version

Versions 7.0 and older of the IEC 61970-301 standard used the serialization format:

```
<cim:ClsName rdf:ID="_ABEB635F-729D-24BF-B8A4-E2EF268D8B9E">
  <cim:ClsName.Assoc rdf:resource="#_73C512BD-7249-4F50-50DA-D93849B89C43"/>
</cim:ClsName>
```

Version 8.0 of the standard has changed the serialization format to specify that the serialization identifier must be a UUID:

```
<cim:ClsName rdf:about="urn:uuid:abeb635f-729d-24bf-b8a4-e2ef268d8b9e">
  <cim:ClsName.Assoc rdf:resource="urn:uuid:73c512bd-7249-4f50-50da-d93849b89c43"/>
</cim:ClsName>
```

To enable CIMantic Graphs to handle different serializations dynamically, the serialization version can be passed as an integer argument:

[4]:
```
params = ConnectionParameters(iec61970_301 = 7) # Use with rdf:ID
```

[5]:
```
params = ConnectionParameters(iec61970_301 = 7) # Use with rdf:about:urn:uuid
```

2.2 2. Creating a Connection

CIMantic Graphs currently supports the following data sources:

- Blazegraph Database
- GraphDB Database
- Neo4J Database
- MySQL Database (in progress)
- GridAPPS-D Platform
- AVEVA PI Asset Framework (in progress)
- RDFLib File Parser

To create the database connection, specify the url / host+port and username/password based on the specific requirements of each database connection driver. Examples of connection parameters for the supported databases are listed below.

2.3 3. Blazegraph Database

Blazegraph DB is a ultra high-performance open-source database supporting Blueprints and RDF/SPARQL APIs. It supports up to 50 Billion edges on a single machine.

Docker images preloaded with IEEE test feeders are available from https://hub.docker.com/r/gridappsd/blazegraph/tags.

To connect to Blazegraph, import the `BlazegraphConnection` class:

[6]:
```
# Import class from cimgraph databases module
from cimgraph.databases.blazegraph import BlazegraphConnection
# Create connection parameters
params = ConnectionParameters(url = "http://localhost:8889/bigdata/namespace/kb/sparql
↪",
                              cim_profile='rc4_2021', iec61970_301=8)
# Create database connection object
blazegraph = BlazegraphConnection(params)
```

2.4 4. Graph DB Database Connection

GraphDB is a full-featured commercial RDF graph database developed by Ontotext

[7]:
```
# Import class from cimgraph databases module
from cimgraph.databases.graphdb import GraphDBConnection
# Create connection parameters
params = ConnectionParameters(url = "http://localhost:7200/repositories/cim_test",
                              cim_profile='rc4_2021', iec61970_301=8)
# Create database connection object
graphdb = GraphDBConnection(params)
```

2.5 5. Neo4j Database Connection

[8]:
```
# Import class from cimgraph databases module
from cimgraph.databases.neo4j import Neo4jConnection
# Create connection parameters

params = ConnectionParameters(url = "neo4j://localhost:7687/neo4j", database="neo4j",
                               cim_profile='rc4_2021', iec61970_301=8)
# Create database connection object
neo4j = Neo4jConnection(params)
```

2.6 6. MySQL-JSON-LD Database Connection

MySQL database with JSON-LD typing generated by the PNNL CIM-Loader library

[ ]:
```
# Import class from cimgraph databases module
from cimgraph.databases.mysql.mysql import MySQLJSONConnection
# Create connection parameters
params = ConnectionParameters(host= "localhost", database="rc4_2021", username="root",
↪ password="password",
                              cim_profile='rc4_2021', namespace="http://iec.ch/TC57/
↪CIM100#")

# Create database connection object
mysql = MySQLJSONConnection(params)
```

2.7 GridAPPS-D Platform Connection

[ ]:
```
# Import class from cimgraph databases module
from cimgraph.databases.gridappsd import GridAPPSDConnection
# Create connection parameters
params = ConnectionParameters(host= "localhost", port="61613", username="app_user",␣
↪password="1234App",
                              cim_profile='rc4_2021', namespace="http://iec.ch/TC57/
↪CIM100#")
# Create platform connection object
gapps = GridAPPSDConnection(params)
```

2.8 AVEVA PI Asset Framework (PI-Web-API) Connection

[ ]:
```
#TODO
```

[ ]:
```

```

2.9 Local Connection (No Database)

[ ]:
```
#TODO
```

[ ]:
```

```

2.10 RDFLib File Parser Connection (No Database)

[ ]:
```
from cimgraph.databases.rdflib.rdflib import RDFlibConnection
# RDFLib File Reader Connection
params = ConnectionParameters(filename="./maple10bus.xml",
                              cim_profile='rc4_2021', iec61970_301=7)
rdf = RDFlibConnection(params)
```

[ ]:
```

```

CHAPTER

THREE

DISTRIBUTION FEEDER MODELING

3.1 Creating a FeederModel

Import all required libraries for data profile, connection parameters, database, and feeder:

```
[ ]: import cimgraph.data_profile.rc4_2021 as cim
     from cimgraph import ConnectionParameters
     from cimgraph.databases.graphdb import GraphDBConnection
     from cimgraph.databases.blazegraph import BlazegraphConnection
     from cimgraph.models import FeederModel
```

```
[ ]: # GraphDB Connection
     params = ConnectionParameters(url = "http://localhost:7200/repositories/cim_test",
                                   cim_profile='rc4_2021', iec61970_301=8)
     graphdb = GraphDBConnection(params)
```

```
[ ]: # Blazegraph Connection
     params = ConnectionParameters(url = "http://localhost:8889/bigdata/namespace/kb/sparql
     ↪",
                                   cim_profile='rc4_2021', iec61970_301=8)
     blazegraph = BlazegraphConnection(params)
```

Create CIM EquipmentContainer object:

```
[ ]: feeder_mrid = "49AD8E07-3BF9-A4E2-CB8F-C3722F837B62"  # 13 bus
     feeder = cim.Feeder(mRID=feeder_mrid)
```

```
[ ]: network = FeederModel(connection=graphdb, container=feeder, distributed=False)
```

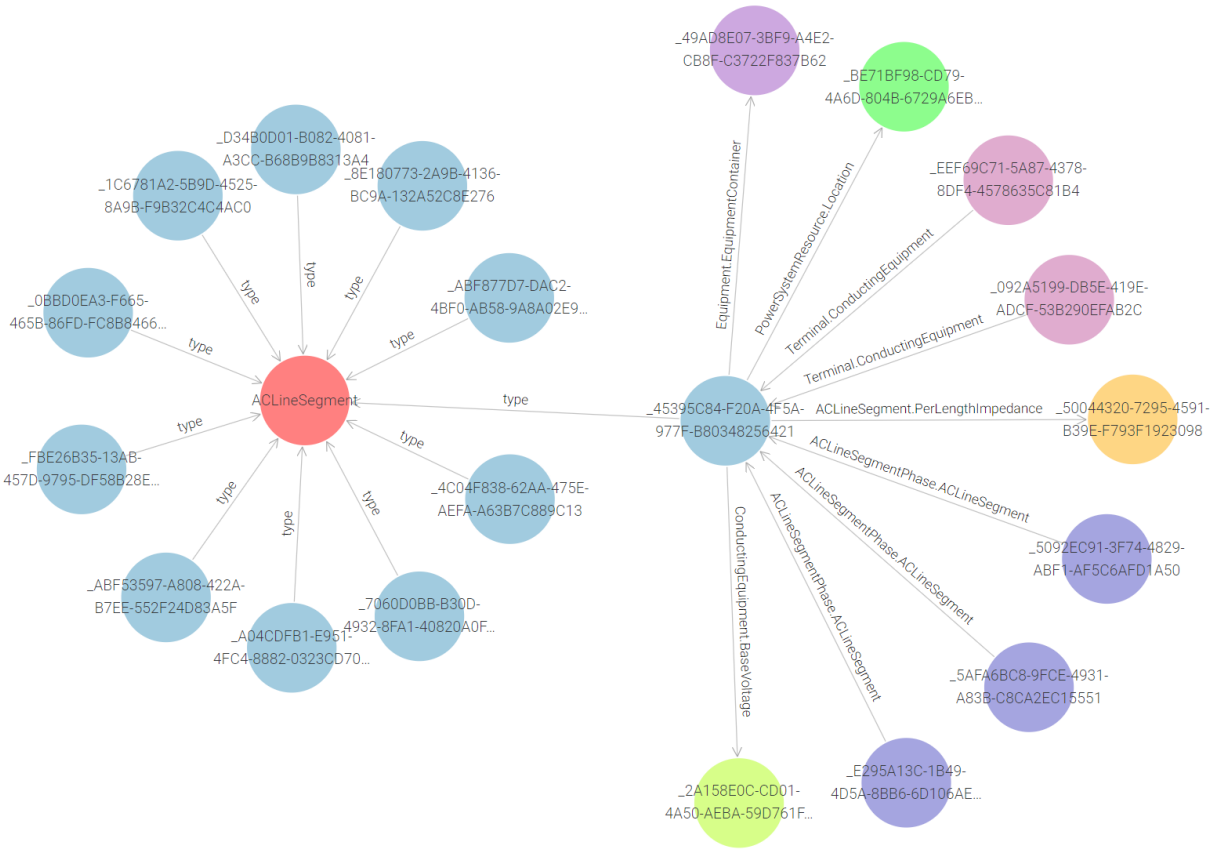
```
[ ]: network = FeederModel(connection=blazegraph, container=feeder, distributed=False)
```

3.2 Traversing the Property Graph

[ ]:
```
network.pprint(cim.ACLineSegment)
```

3.3 Example 1: Expand the Property Graph by One Edge

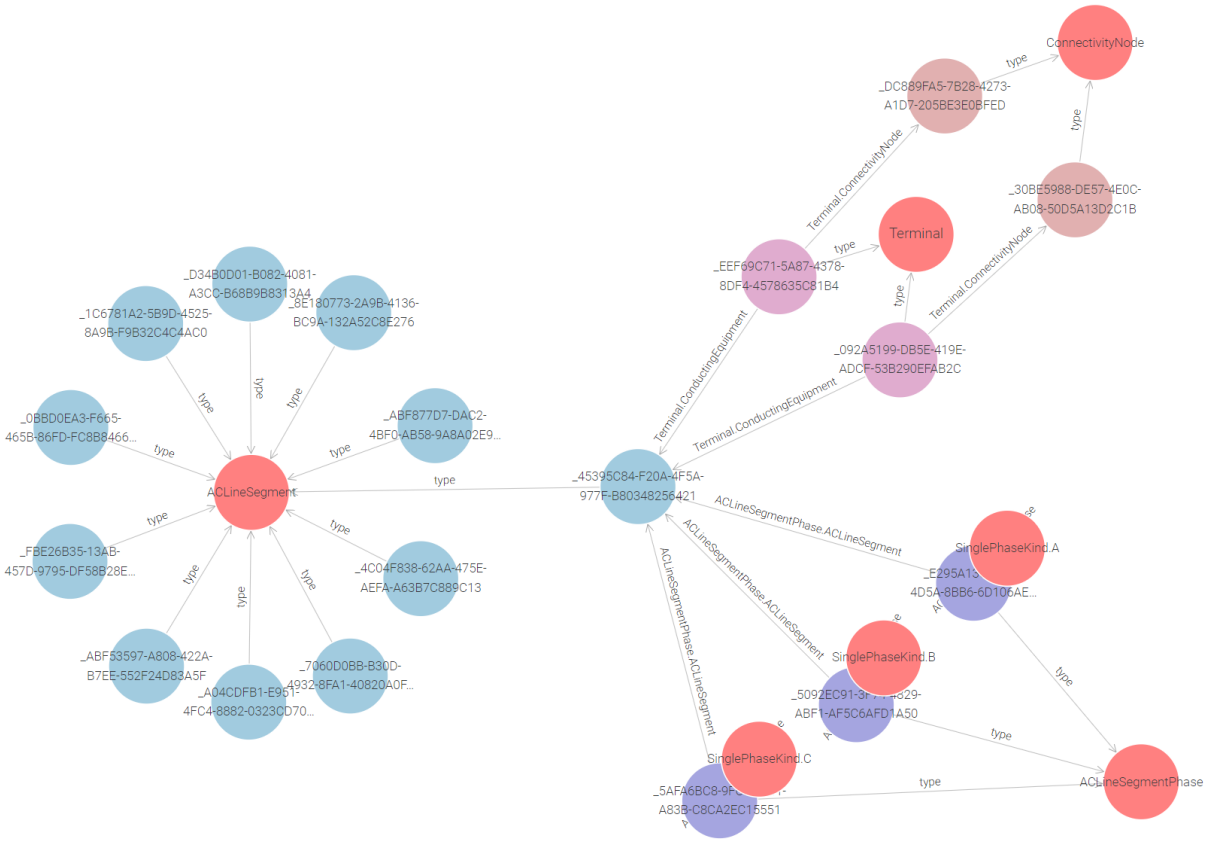
```
[ ]: network.get_all_edges(cim.ACLineSegment)
```




```
[ ]: network.pprint(cim.ACLineSegment)
```

3.4 Example 2: Expand CIM-Graph to Find Bus and Phase of Each Line

```
[ ]: network.get_all_edges(cim.ACLineSegment)
     network.get_all_edges(cim.ACLineSegmentPhase)
     network.get_all_edges(cim.Terminal)
```




```
[ ]: network.pprint(cim.ACLineSegmentPhase)
```

```
[ ]: for line in network.graph[cim.ACLineSegment].values():
         print('\n line mrid: ',line.mRID)
         print('line name:', line.name)
         print('bus 1: ', line.Terminals[0].ConnectivityNode.name)
         print('bus 2: ', line.Terminals[1].ConnectivityNode.name)

         for line_phs in line.ACLineSegmentPhases:
             print('phase:', line_phs.phase, ', sequence:', line_phs.sequenceNumber)
```

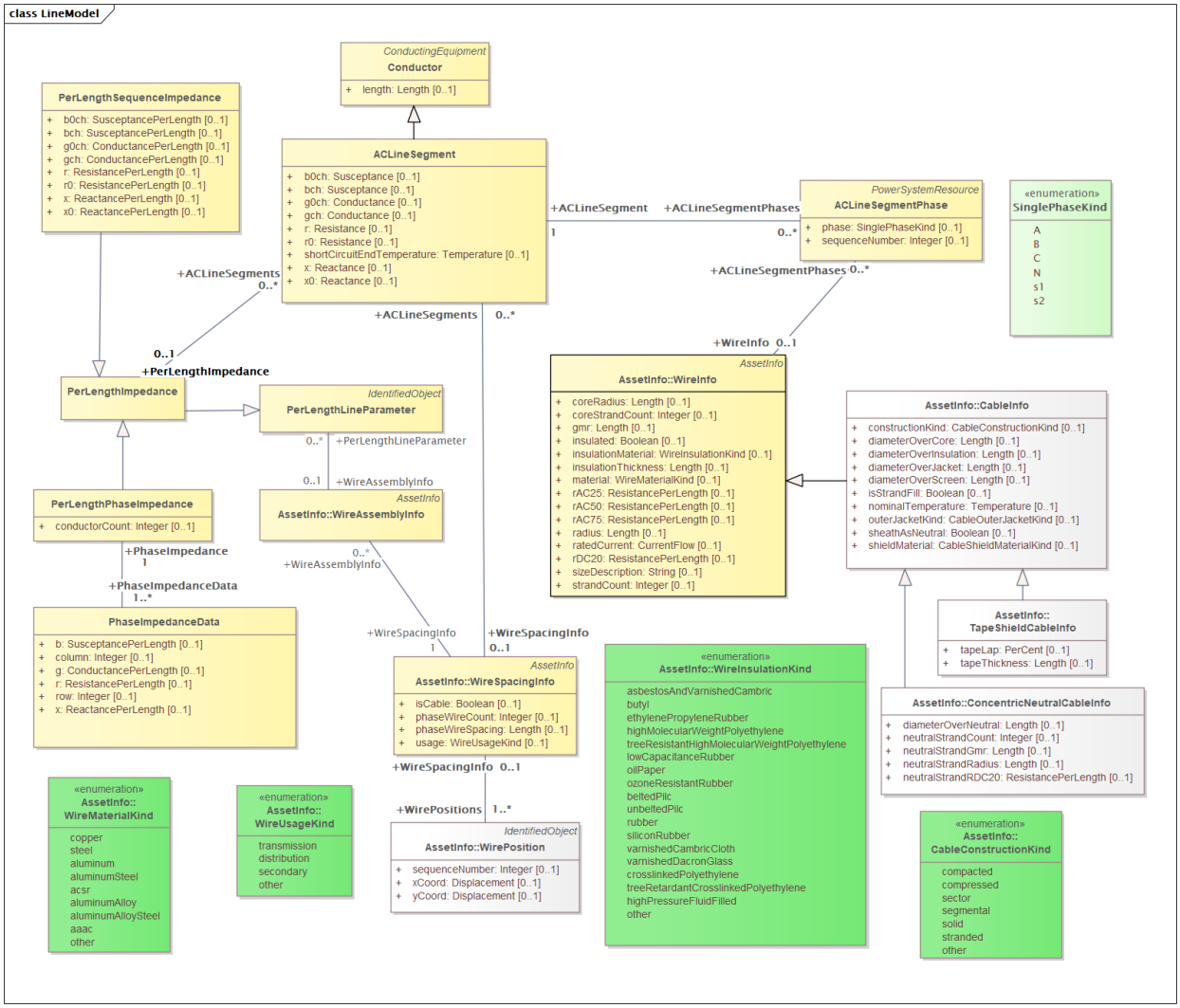
3.5 Example 3: Get All Measurements

All SCADA points are associated in memory with the correct power system objects

```
[ ]: network.get_all_edges(cim.Analog)
     network.get_all_edges(cim.Discrete)
```

```
[ ]: for line in network.graph[cim.ACLineSegment].values():
         for meas in line.Measurements:
             print('Measurement: ', meas.name,  ', type:', meas.measurementType, ', phases:
     ↪', meas.phases)
```

3.6 Example 4: Get all line impedances and physical asset info



[ ]:
```
network.get_all_edges(cim.ACLineSegment)
network.get_all_edges(cim.ACLineSegmentPhase)
network.get_all_edges(cim.PerLengthPhaseImpedance)
network.get_all_edges(cim.PhaseImpedanceData)
network.get_all_edges(cim.WireSpacingInfo)
network.get_all_edges(cim.WirePosition)
network.get_all_edges(cim.OverheadWireInfo)
network.get_all_edges(cim.ConcentricNeutralCableInfo)
network.get_all_edges(cim.TapeShieldCableInfo)
network.get_all_edges(cim.Terminal)
```

3.6.1 Example 4.1: Parse by PSR:

[ ]:
```
for line in network.graph[cim.ACLineSegment].values():
    print('\n line mrid: ', line.mRID)
    print('line name:', line.name)

    for line_phs in line.ACLineSegmentPhases:
        print('phase ', line_phs.phase, ': ', line_phs.mRID)
        if line_phs.WireInfo is not None:
            print('type: ', line_phs.WireInfo.__class__.__name__)
            print('gmr: ', line_phs.WireInfo.gmr)
            print('insulated: ', line_phs.WireInfo.insulated)

    if line.WireSpacingInfo is not None:
        for position in line.WireSpacingInfo.WirePositions:
            print('seq:', position.sequenceNumber, ' x:', position.xCoord, ' y:',␣
↪position.yCoord)

    if line.PerLengthImpedance is not None:
        for data in line.PerLengthImpedance.PhaseImpedanceData:
            print('row:', data.row, 'col:', data.column, 'r:', data.r, 'x:', data.x,
↪'b:', data.b)
```

3.6.2 Example 4.2: Parse by Asset

[ ]:
```
for impedance in network.graph[cim.PerLengthPhaseImpedance].values():
    print('\n name:', impedance.name)
    for data in impedance.PhaseImpedanceData:
            print('row:', data.row, 'col:', data.column, 'r:', data.r, 'x:', data.x,
↪'b:', data.b)
    for line in impedance.ACLineSegments:
        node1 = line.Terminals[0].ConnectivityNode
        node2 = line.Terminals[1].ConnectivityNode
        print('Line: ', line.name)
        print('Buses:', node1.name, node2.name)
```

[ ]:
```
for line in network.graph[cim.ACLineSegment].values():
    for meas in line.Measurements:
        print('Measurement: ', meas.name,  ', type:', meas.measurementType, ', phases:
↪', meas.phases)
```

[ ]:
```

```

CHAPTER

FOUR

BUS-BRANCH TRANSMISSION MODELING

4.1 Creating a BusBranchModel

Import all required libraries for data profile, connection parameters, database, and `BusBranchModel`:

```
[ ]: import cimgraph.data_profile.rc4_2021 as cim
     from cimgraph import ConnectionParameters
     from cimgraph.databases.blazegraph import BlazegraphConnection
     from cimgraph.models import BusBranchModel
```

```
[ ]: # Blazegraph Connection
     params = ConnectionParameters(url = "http://localhost:8889/bigdata/namespace/kb/sparql
     ↪",
                                   cim_profile='rc4_2021', iec61970_301=8)
     blazegraph = BlazegraphConnection(params)
```

Create CIM EquipmentContainer object:

```
[ ]: model_mrid = "1783D2A8-1204-4781-A0B4-7A73A2FA6038" #IEEE 118 Bus
     container = cim.ConnectivityNodeContainer(mRID=model_mrid)
```

```
[ ]: network = BusBranchModel(connection=blazegraph, container=container,␣
     ↪distributed=False)
```

```
[ ]: network.get_all_edges(cim.ACLineSegment)
     network.pprint(cim.ACLineSegment)
```

```
[ ]:
```