Seminar - Energy Informatics

Automatic Generation of Power Systems in Simulink

By
Bernhard Krop

1. Introduction

Nowadays, **electrical power systems** are everywhere. There are small systems, like the cables, switches and lights in a house. They are easy to overlook and understand. But there are some very large power systems, like electrical systems of cables, transformators and generators, which are connected together over the whole world. They are not easy to understand and definitely not easy to overlook.

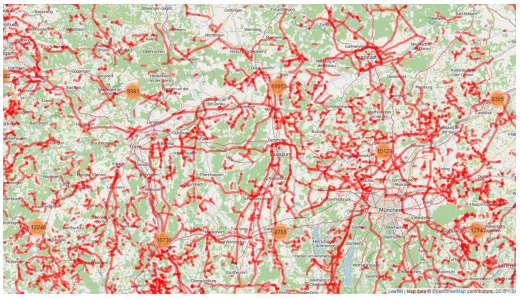

Figure 1: An extract of OpenGridMap. It shows several points and the connections between them.[1]

Nevertheless, platforms, which collects and stores the data of such large power systems, exists. An example of these platforms is **OpenGridMap**, which describes the electrical system of the entire world. It stores the position, the operator, contact informations of the operator and some other data of points, like what kind of electrical device it is. For further reading about OpenGridMap is the paper of the 6th '**Conference on Energy Informatics**'[16] recommended.

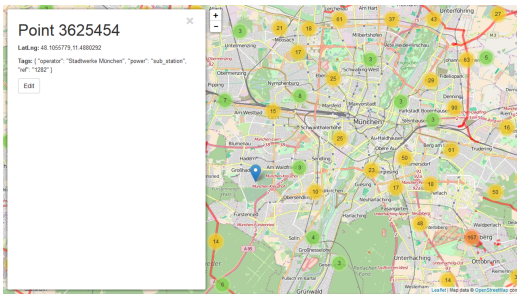

Figure 2: This extract of OpenGridMap shows several points. The single (blue) point is selected.[1]

This stored data is used to improve the infrastructure of existing power systems. But to know, whether it is an improvement, some **simulations** are necessary. There occures the first problem: If someone wants to simulate something, it has to be build or created in a simulator. To create such large power systems per hand would be an incredible task. Therefor, that power systems should be created automatically. And there occures the second problem: How should the stored data be brought to such a creator? The answer to this question is the '**Common Information Model**'.

2. The Common Information Model

The '**Common Information Model**' (**CIM**)[8] is a standard, developed by the electric power industry and officially adopted by the '**International Electrotechnical Commission**' (**IEC**)[2]. The aim of CIM is to exchange information about electrical networks. It's maintained as an **UML** model (chapter 2.1), which includes several packages. Each package contains even more packages, diagrams or descriptions of classes, enumerations, etc.

Figure 3: The Logo of the International Electrotechnical Commission.[2]

Actually, CIM contains more than just packages for electrical systems. It is expended to define completely other systems, for example markets or economies, too. For power systems, the relevant package is '**Electricity Domain Model/TC57CIM/IEC61970**'. For more or more detailed informations about CIM, please, visit the website of the '**Carbon Credits Market Model**'[3]. There is the most recent version of the Common Information Model.

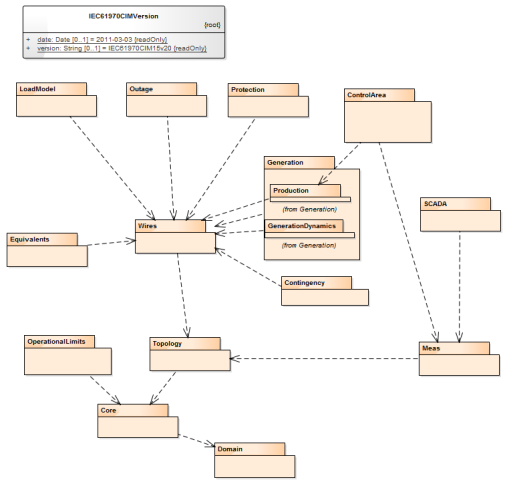

Figure 4: The main diagram of the CIM package IEC61970.[3]

As we will see in chapter 2.1, UML is not really good for practical uses in informatics. That's why some '*design artifacts*' are derived from CIM, to work with them instead of the UML diagrams. Several of these derivatives exists, but in this seminar work, only **XML** (chapter 2.2) and **RDF** (chapter 2.3) will be shown. They store the whole network - modeled in CIM - in a simple **text file**, which is easier to work with.

2.1 The Unified Modeling Language

The '**Unified Modeling Language**' (**UML**)[13] is a modeling language in **software engineering** with the intention to design systems. It includes several kinds of diagrams, which are separated in structural and behavioral diagrams. Because CIM is modeled as a **class diagram**, which is a structural diagram, only this part of UML is shown here.

A class diagram[7] describes the structure of a system by showing the system's **classes**, their **attributes**, **methods** and the **relationships** between them. Every system, modeled in a class diagram, is abstract. That means, that there is no instance of one of the classes. To describe a system with it's real objects at a specific time, **object diagrams**[10] are used.

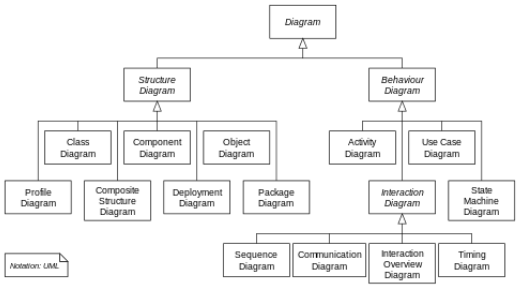

Figure 5: All sorts of diagrams, which are contained in UML.[13]

The following informations about class diagrams - and some more - can all be found on Wikipedia[7].

- **Classes**

  Classes are represented as a simple rectangle with up to three fields. The upper field contains the name of the class, the middle field contains the attributes of the class and the lower field contains the methods of the class. Attributes have types, while methods can - but needn't - have parameters and a type of return. Both, attributes and methods have a modifier for their visibility. No modifier symbol equals the modifier for the default visibility.

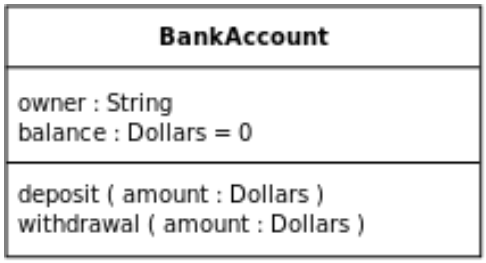

Figure 6: The class 'BankAccount'.[7]

The figure above shows the class 'BankAccount'. This class has two attributes called 'owner' (of type 'String') and 'balance' (of type 'Dollars'). Additionally, the attribute 'balance' has a default value, which is zero. It has also two methods: 'deposit' and 'withdrawal'. 'deposit' needs a parameter called 'amount' of type 'Dollars' and returns nothing. 'withdrawal' needs a parameter called 'amount' of type 'Dollars' and returns nothing, too. All attributes and methods have default visibility.

- **Associations**

Associations between two classes are relationships between these classes, in which the first class is an attribute of the second class and vice versa. They are visualized as a line between the classes, which are part of this association. The association has on both ends the name of the attribute, its visibility modifier and the multiplicity of them.

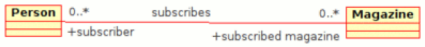

Figure 7: This figure shows an association 'subscribes' between two classes 'Person' and 'Magazine'.[7]

The figure above shows two classes: 'Person' and 'Magazine'. This classes are in relationship by the association 'subscribes'. Each person has the attribute 'subscribed magazine', which contains all magazines, which the person has subscribed. Each magazine has the attribute 'subscriber', which contains all persons, who has subscribed to this magazine. The mulitpliers '0..*' means, that each person can subscribe to any number of magazines and each magazine can have any number of subscribers.

- **Aggregations**

An aggregation is a 'has'-relationship, where one class has or is part of another one.

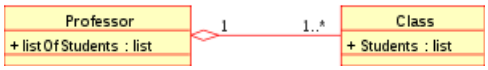

Figure 8: Here is shown an aggregation between the classes 'Professor' and 'Class'.[7]

The figure above shows the classes 'Professor' and 'Class'. 'Professor' has a public attribute called 'listOfStudents' of type 'list'. 'Class' has a public attribute called 'students' of type 'list'. The aggregation shows, that every professor has at least one class, while each class is part of exactly one professor.

- **Compositions**

A composition is a stronger variant of an aggregation. The class, which is part of another class, might not exist without this other class.

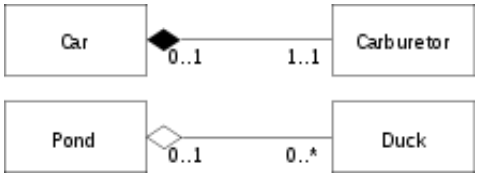

Figure 9: Here are shown a composition and an aggregation, for comparability.[7]

In the figure above are two relationships illustrated. The lower relationship is an aggregation between the classes 'Pond' and 'Duck'. Each pond can have any number of ducks, while every duck can be part of one or no pond. The upper relationship is a composition between the classes 'Car' and 'Carburetor'. Each car has exactly one carburetor, while a carburetor can be part of one or no car (e.g. if the carburetor is a spare part). But if a carburetor is part of a car, it depends on it. For example, if the car get destroyed, its carburetor is probably destroyed, too.

- **Generalization**

A generalization indicates, that one class (the subclass) is a specialization of another class (the superclass). Conversely is the superclass a generalization of the subclass.

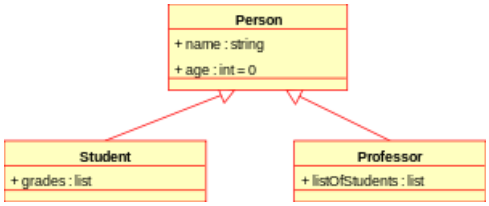

Figure 10: An example about generalization.[7]

The figure above shows three classes: 'Person', 'Student' and 'Professor'. Every person has a name and an age; every student has a list of grades and every professor has a list of students. But students and professors are persons, too. So every student has a name and an age as well as every professor. However, a professor is not a student and has no grades, while a student is no professor and has no list of students.

2.2 The Extensible Markup Language

The '**Extensible Markup Language**' (**XML**)[15] is a markup language, which brings documents in a format, which is readable for humans as well as for machines. It is defined by the '**W3C's XML 1.0 Specification**'[4] and some other specifications, which are all free open standards. By definition, a XML document is a string of unicode characters and the file ending is '**.xml**'.

- **Tags**

  A Tag begins with the symbol '<' and ends with the symbol '>'. Three kinds of tags exists: a **start-tag** (e.g. <section>), an **end-tag** (e.g. </section>) and **empty-element tags** (e.g. <line-break />).

- **Elements**

  An element either begins with a start-tag and ends with an end-tag or is an empty-element tag. The content of the element, if it has one, is between the start- and end-tag. Examples are <Greeting>Hello, world.</Greeting> or <line-break />.

- **Attributes**

  Elements can have varies attributes. Each attribute can appear at most once and can have only one value. They consists of name-value pairs and are either within a start-tag or an empty-element tag. In the example <step number="3">Connect A to B.</step> is 'number' an attribute of the tag 'step' and its value is '3'.

- **Comments**

  A comment in XML starts with '<!–' and ends with '–>'. So that '<!– Hello, world. –>' is a comment.

- **Declaration**

  Every XML document begins with a declaration, which has some informations about XML itself. An example would be <?xml version="1.0" encoding="UTF-8"?>.

The following **example** shows how to use XML correctly.

```
<?xml version = "1.0" encoding = "UTF - 8"? >
<! - - ********** - - >
<! - - Some cinema films. - - >
<! - - ********** - - >
< films >
  < film >
    < title > Star Wars VII < /title >
    < genre > Science - Fiction < /genre >
  < /film >
  < film >
    < title > ThePeanuts < /title >
    < genre > Animation < /genre >
  < /film >
  < film >
    < title > Heidi < /title >
    < genre > Children < /genre >
  < /film >
< /films >
<! - - ********** - - >
<! - - Some video games. - - >
<! - - ********** - - >
< games >
  < game >
    < title > The Elder Scrolls III < /title >
    < subtitle > Morrowind < /subtitle >
    < players > Singleplayer < /players >
  < /game >
  < game >
    < title > Minecraft < /title >
    < players > Multiplayer < /players >
  < /game >
  < game >
    < title > League of Legends < /title >
    < players > Multiplayer < /players >
```

```
    </game>
    <game>
      <title>The Witcher III</title>
      <subtitle>Wild Hunt</subtitle>
      <players>Singleplayer</players>
    </game>
  </games>
```

2.3 The Resource Description Framework

The '**Resource Description Framework**' (**RDF**)[11] is an abstract model, which is defined by the '**W3C's Specifications**'[5]. It is used for modeling of information which are implemented in web resources.

RDF is based upon the idea of making statements about resources in the form of **subject-predicate-object** expressions, also called '**triples**'. The subject denotes the resource and the predicate denotes the relationship between subject and object. A little example would be the sentence "The sky has the color blue.". "The sky" is the subject, "has" is the predicate and "the color blue" is the object.

The subject is either a '**Uniform Resource Identifier**' (**URI**)[14] (a string to identifiy the name of a resource) or a blank node. The predicate is a URI, which represents a relationship. The object is a URI, blank node or a string. Even if RDF is modeled for accessing data on the Internet, by using an URL as URI, it is not limited to that. If an application knows the structure of a scheme, it doesn't need to use the World Wide Web, even if the URI begins with 'http'.

As mentioned before, RDF is an abstract model. It has variant formats and implementations. The first one - and that one, that will introduced here - is **RDF/XML**[5] (often misscalled RDF). This format is used to combine the syntax of XML with the main idea of RDF.

RDF/XML uses, like XML, an own header for RDF. This line specifies the used syntax of RDF and the schemes, which are in use in the document. A minimal example would be "<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">", which makes use of the RDF syntax, defined by the W3C's specification of 1999. This header is a start-tag and needs an end-tag, which is "</rdf:RDF>".

Let assume, we want to describe the contact datas of a person called "Erich Miller". His e-mail address is "e.miller123@example" and his title is "Dr.". By using the scheme for contact data of persons, we can write the following RDF/XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:contact="http://
  www.w3.org/2000/10/swap/pim/contact#"
  xmlns:eric="http://www.w3.org/People/
  EM/contact#" xmlns:rdf="http://
  www.w3.org/1999/02/
  22-rdf-syntax-ns#">
<rdf:Description rdf:about="http://
  www.w3.org/People/EM/contact#me">
    <contact:fullName>Erich Miller</
      contact:fullName>
</rdf:Description>
<rdf:Description rdf:about="http://
  www.w3.org/People/EM/contact#me">
    <contact:mailbox rdf:resource="mailto:
      e.miller123@example"/>
</rdf:Description>
<rdf:Description rdf:about="http://
  www.w3.org/People/EM/contact#me">
    <contact:personalTitle>Dr.</contact:
      personalTitle>
</rdf:Description>
<rdf:Description rdf:about="http://
  www.w3.org/People/EM/contact#me">
    <rdf:type rdf:resource="http://
      www.w3.org/2000/10/swap/pim/
      contact#Person"/>
</rdf:Description>
</rdf:RDF>
```

2.4 CIM in RDF/XML

Now, we know all we need to understand the definition of a **CIM** class and can it instantiate in **RDF/XML** by using the corresponding CIM scheme. We use the definitions of the mentioned package of the '**Carbon Credits Market Model**'[3]. For a simple example, go to chapter 5, please.

3. The Simulation Environment

Now, we know everthing we need to understand the **Common Information Model**[8]. The second question, which was questioned at the beginning (chapter 1), was about simulations. As environment was choosen **Simulink** (chapter 3.2), an extension of **MATLAB** (chapter 3.1). Let's take a closer look at them.

3.1 MATLAB

MATLAB[9] is a highly developed language and interactive environment, which is used of millions of developers and researchers in the entire world. It is developed by **MathWorks**[6]. Primarily, MATLAB is a **functional programming language**. Every line of code is evaluated immediately. Let's take a look at the most basic syntax of MATLAB.

- **Variables**

  The operator '=' is used to define a variable in MATLAB. A variable is defined only once. Their value cannot be changed. But a variable can reassigned. In that case, the old value of the variable is lost. For example:

```
>> x = 0
x =
  0
>> word = 'Hello'
word =
  'Hello'
>> x = 'World'
x =
  'World'
```

- **Vectors and Matrizes**

  Vectors and Matrizes in MATLAB are simple arrays. There are various ways to define or create an array. The simplest one is to use square brackets - '[' and ']' - and the separating symbol ';'. To get the value at a specific position in an array, the brackets '(' and ')' are used.

```
>> array = [1 2 3; 4 5 6; 7 8 9]
array =
  1 2 3
  4 5 6
  7 8 9
>> array(2, 1)
ans =
  4
```

3.2 Simulink

Simulink[12] is an extension to MATLAB and a **graphical programming environment** for modeling, simulating and analyzing multidomain dynamic systems, which is also developed by **MathWorks**[6]. Models in Simulink are build by using several blocks and connect them. Some of the most common blocks are introduced below.

- **Constant**

  The constant block generates a constant value and has one output.

Figure 11: The Constant block.

- **Inport**

  The inport block is used to provide an input port for subsystems or an external input.

Figure 12: The Inport block.

- **Outport**

  The outport block is used to provide an output port for subsystems or an external output.

Figure 13: The Outport block.

- **Subsystem**

  The subsystem block is a block, which contains a whole model. With inport and outport blocks can this model be accessed.

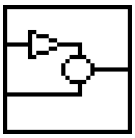

Figure 14: The Subsystem block.

Of course, it is possible to manipulate a model in Simulink with MATLAB code. Some of these functions will now be introduced.

- **new_system**

  The function 'new_system' creates a new Simulink model. This model will not be shown. It just exists and can be manipulated.

- **open_system**

  The function 'open_system' opens an existing Simulink model and shows it in a new window.

- **close_system**

  The function 'close_system' closes an existing Simulink model. After closing, this model cannot be manipulated anymore.

- **add_block**

  The function 'add_block' adds a new block to a Simulink model.

- **delete_block**

  The function 'delete_block' deletes an existing block of a Simulink model.

- **set_param**

  The function 'set_param' sets a parameter of a block in a Simulink model.

- **get_param**

  The function 'get_param' returns a parameter of a block in a Simulink model.

- **add_line**

  The function 'add_line' adds a new line from one block to another in a Simulink model.

- **delete_line**

  The function 'delete_line' deletes an existing line of a Simulink model.

4. Generate Power Systems

After we learned everything we must know about the **simulation environment** and the **theoretical knowlegde**, we can start to look over the script I wrote.

I started some considerations and made the conclusion, that I should break the work into several tasks. I decided to split it into **three parts**: read a CIM file and parse the objects; create the Simulink model with the parsed objects; an interface for the user, which make the subcalls and shows the created model. The scripts are called (in the same order as before) '**parseCIM.m**', '**createSystem.m**' and '**generatePowerSystem.m**'.

4.1 generatePowerSystem.m

The file 'generatePowerSystem.m' is the **entry point** of the whole program and the interface for the user. It just checks the input paramters and make subcalls. Therefore, it is a very small script. The equally named function has the following declaration:

$$system = generatePowerSystem(path,\ title)$$

Both parameters are **optional**. '**path**' is the path for the CIM file. If it doesn't exist, "Input.txt" is assumed. '**title**' will be the title of the Simulink model. If it doesn't exist, the model gets the title "untitled". '**system**' is the **return value** of this function. It holds the handle of the Simulink model.

After checking, that the existing input parameters are correct, the function calls the script 'parseCIM.m', which parses all CIM objects in a string array. Afterwards, it calls 'createSystem.m' and let it create the Simulink model. If that is done, too, the last thing left to do is just to **open the system**.

4.2 parseCIM.m

The file 'parseCIM.m' does the **most important work**. Without the CIM objects, which it parses, the whole Simulink model could not be created. But for that importance, this file contains not much code. Because it needn't to know what objects it parses. It just parses some objects. The identification of this objects is done in the file 'createSystem.m'.

The function has the following declaration:

$$parseCIM()$$

The only parameter needed in this function is the **path of the input file**. That is the parameter 'path' in 'generatePowerSystem' and is made **globally**. That means, that it has not to give further to 'parseCIM' as parameter. The same counts for the return value. Because it will made globally, it **need not to get returned**. That is the reason, why this function has neither parameters nor a return value.

After opening it, 'parseCIM' reads the whole input file. Let's take a closer look into the code.

$$l\_sData = strtrim(fgetl(l\_iFileID));$$

This single line of code **reads the next line** of the input file and stores it into the variable called 'l_sData' ('l' for local, 's' for string, 'Data' is the real name of the variable). '**fgetl**' reads the next line of a file. The file ID is stored in 'l_iFileID' (i for integer). The function '**strtrim**' removes any white spaces at the beginning and end of the parsed string. At the end, l_sData contains a, maybe empty, string.

```
l_aCommentStart = strfind(l_sData,
  '<!--');
l_cSize = size(l_aCommentStart);
while(l_cSize > 0)
  l_aCommentEnd = strfind(l_sData,
    '-->');
  while(size(l_aCommentEnd) <= 0)
    l_sData = strcat(l_sData,
      strtrim(fgetl(l_iFileID)));
    l_aCommentEnd = strfind(l_sData,
      '-->');
  end
  l_sData = strcat(l_sData(1 :
    l_aCommentStart(l_cSize(1 , 2)) - 1),
    l_sData(l_aCommentEnd(1) + 3 : end));
  l_aCommentStart = strfind(l_sData,
    '<!--');
  l_cSize = size(l_aCommentStart);
end
```

This code snippet removes any XML comments (chapter 2.2) which might be in l_sData.

'**strfind**' searches through the string in its first parameter for the string in its second parameter. Because we want to **find and remove comments**, we search for the string '<!–' in l_sData. The result of this search will be stored in 'l_aCommentStart' ('a' for array). The return value of strfind is an array, because it returns the index positions of the occurrences of the second parameter in the first parameter. If the size is not greater than zero, there are no comments and we are done. Else, we have found the start-tag of an XML comment and must now find the end-tag '–>'. As before, we use strfind, but now, we store the result in 'l_aCommentEnd'. If the end-tag is not in l_sData, it was not read, yet. In that case, we must concatenate l_sData and the next line of the input file, until the end-tag was found. Now, we can delete the comment by **concatenate** the string before the start-tag and the string after the end-tag and store the result again in l_sData. We repeat this for every line, until **no comments are left**.

```
l_cSize = size(l_sData);
if((l_cSize(1) <= 0) || (l_cSize(2) <= 0))
  continue;
end
```

'size' doesn't return a number, it returns a cell with the sizes of all dimensions of its parameter. By just using it, we use the value of the first dimension. If we want to use another one, we have to store it in a variable. That's why we use 'l_cSize' ('c' for cell). The first dimension of l_sData should be one (a string) or zero (no string or NULL). The second dimension is the length of the string. If one of these dimensions is less or equal to zero, we have **an empty or no string** and can **skip it**.

Now, the string in l_sData contains an **XML object**. But we are only interested in CIM objects. By using **switch cases**, we can filter them and ignore the others.

```
l_cSeparators = strfind(l_sData, ' ');
if(size(l_cSeparators) <= 0)
  warning(...);
  continue;
end
l_sTag = l_sData(2 : l_cSeparators(1, 1) -
  1);
l_cTagEnd = strfind(l_sData, strcat('</',
  l_sTag, '>'));
while(size(l_cTagEnd) <= 0)
  l_sData = strcat(l_sData,
    strtrim(fgetl(l_iFileID)));
  l_cTagEnd = strfind(l_sData, strcat('</',
    l_sTag, '>'));
end
```

This code snippet above takes the **whole CIM object**. From the start-tag to the end-tag, the entire string will be stored in l_sData. As first, we just have the **start-tag** and must find the **end-tag**. Therefore, we have to identifiy the name of this object. That's pretty simple, because we know, that after the name of the tag is a space. If there is no space in l_sData, we just throw a warning and ignore the read data. Now, we store the tag into 'l_sTag' and search for the end-tag by using strfind. The result is stored in 'l_cTagEnd'. It can be, that the end-tag is not read, yet. For that case, we just have to read more lines. After the end-tag is found, l_sData contains the whole CIM object.

```
l_cQuoteChars = strfind(l_sData, '"');
if(size(l_cQuoteChars) <= 1)
  warning(...);
  continue;
end
l_sID = l_sData(l_cQuoteChars(1) + 1 :
  l_cQuoteChars(2) - 1);
l_cTagStart = strfind(l_sData, '<');
l_cTagEnd = strfind(l_sData, '>');
l_cSize = size(l_cTagStart);
l_sData = l_sData(l_cTagEnd(1) + 1 :
  l_cTagStart(l_cSize(2)) - 1);
g_cObjects = cat(1, g_cObjects, {l_sTag(5 :
  end), l_sID, l_sData});
```

This code above is the last for the function 'parseCIM'. It **stores the CIM object** into a global variable by concatenate it vertically on a cell. This **global variable**, called 'g_cObjects' ('g' for global), will contain all the CIM objects, which could be parsed. It's a nx3-cell, where n is the number of objects. The first coloumn contains the **name** of the CIM object, the second coloumn contains the **RDF-ID** of the CIM object and the third coloumn contains the **attributes** of the CIM object. The first thing to do here is to store the RDF-ID into a variable. Because it is between two quotes, we just search for them and store the ID in 'l_sID'. The name

of the tag is simple, because we stored it already in l_sTag in the code before. The last thing what we have to do is to find the attributes. They are between start- and end-tag. By cutting off the start- and end-tag from l_sData, it contains only the attributes. Yet, what we store in g_cObjects is l_sTag, l_sID and l_sData (in that order). and the object is parseed. In that way, **all CIM objects** get parsed and stored in g_cObjects.

4.3 createSystem.m

This file contains the **most and largest work** of the whole program (**more than 4.000 lines of code!**). Because the code is different for each CIM class, the functions, which creates the objects, will not discussed here and only called as **create-functions**.

$$createSystem()$$

Like 'parseCIM', 'createSystem' needs no parameters and no return value, because all necessary variables are global. All that this function does, is to **create a new system** and **add and connect blocks** by calling some create-functions. Which function get called depends on the class of the CIM object. For example, to create a BaseVoltage, the function would be call 'createBaseVoltage'.

5. Example

Consider the following **simple example**. A GeographicalRegion called 'Region' contains one SubGeographicalRegion called 'SubRegion', which contains a Substation called 'Substation'. This substation contains a VoltageLevel called 'Level' and that contains an AC BaseVoltage of 10kV. The CIM file, named 'example.cim', would look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/
  1999/02/22-rdf-syntax-ns#" xmlns:cim=
  "http://iec.ch/TC57/2008/
  CIM-schema-cim14#">
  <cim:BaseVoltage rdf:ID=
    "_Example_Voltage_">
      <cim:IdentifiedObject.name>Voltage</
        cim:IdentifiedObject.name>
      <cim:BaseVoltage.isDC>false</
        cim:BaseVoltage.isDC>
      <cim:BaseVoltage.nominalVoltage>10000
        </cim:BaseVoltage.nominalVoltage>
  </cim:BaseVoltage>
  <cim:GeographicalRegion rdf:ID=
    "_Example_Region_">
      <cim:IdentifiedObject.name>Region</
        cim:IdentifiedObject>
  </cim:GeographicalRegion>
  <cim:SubGeographicalRegion rdf:ID=
    "_Example_SubRegion_">
      <cim:IdentifiedObject.name>SubRegion
        </cim:IdentifiedObject.name>
      <cim:SubGeographicalRegion.Region
        rdf:resource="_Example_Region_"/>
  </cim:SubGeographicalRegion>
  <cim:Substation rdf:ID=
    "_Example_Substation_">
      <cim:IdentifiedObject.name>Substation
        </cim:IdentifiedObject.name>
      <cim:Substation.Region rdf:resource=
        "_Example_SubRegion_"/>
  </cim:Substation>
  <cim:VoltageLevel rdf:ID=
    "_Example_Level_">
      <cim:IdentifiedObject.name>Level</
        cim:IdentifiedObject.name>
      <cim:VoltageLevel.Substation
        rdf:resource="_Example_Substation_"/>
```

$$< cim : VoltageLeve.BaseVoltage$$
$$rdf : resource = "\_Example\_Voltage\_" / >$$
$$< /cim : VoltageLevel >$$
$$< /rdf : RDF >$$

Now we can start MATLAB and call 'generatePowerSystem.m'. The file path is "C:/example.cim" and the name of the Simulink model shall be "Example". The call will look like this:

$$>> \ generatePowerSystem('C : /example.cim', 'Example')$$

What now happens, is, that the path will be stored in a global variable. Then, the subcall parseCIM() is made.

parseCIM knows the file path due to the global variable and opens the file. Like in chapter 4.2, the CIM objects will parsed into a global variable. This variable would look like this:

$Names$:

$'BaseVoltage'$
$'GeographicalRegion'$
$'SubGeographicalRegion'$
$'Substation'$
$'VoltageLevel'$

IDs :

$'\_Example\_Voltage\_'$
$'\_Example\_Region\_'$
$'\_Example\_SubRegion\_'$
$'\_Example\_Substation\_'$
$'\_Example\_Level\_'$

$Attributes$:

$' < cim : IdentifiedObject.name > Voltage < /...'$
$' < cim : IdentifiedObject.name > Region < /...'$
$' < cim : IdentifiedObject.name > SubRegion < /...'$
$' < cim : IdentifiedObject.name > Substation < /...'$
$' < cim : IdentifiedObject.name > Level < /...'$

parseCIM has finished and generatePowerSystem is on its turn, again. It stores the title in a global variable and calls createSystem().

The first thing createSystem does is to check whether a system with the same title exists. If yes, it get closed. After that it creates a new system with our title: "Example". Now, it goes through all objects and call the corresponding create-functions. 'createBaseVoltage' creates a BaseVoltage in the model. 'createGeographicalRegion' creates a Subsystem at the same place as the BaseVoltage. 'createSubGeographicalRegion' creates a Subsystem in the Subsystem 'Region'. 'createSubstation' creates a Subsystem in the Subsystem 'SubRegion'. 'createVoltageLevel' creates a Subsystem in the Subsystem 'Substation' and copies the BaseVoltage on the top level into this new Subsystem. All CIM objects are now created. At last, all temporary blocks have to get deleted. The only temporary block is the BaseVoltage at the top level.

Right now, createSystem has finished and we are back in generatePowerSystem. It's only one thing left to do: open the just created system. The figures below are showing this system.

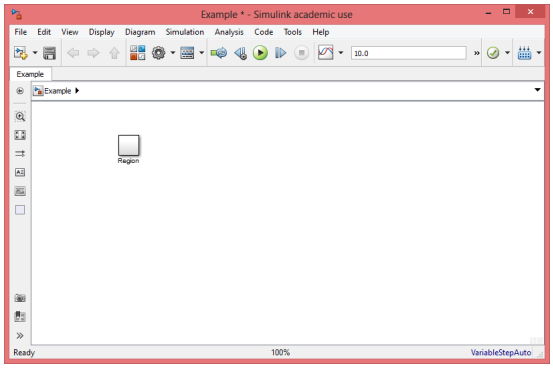

Figure 15: The system "Example", which contains the 'GeographicalRegion' "Region".

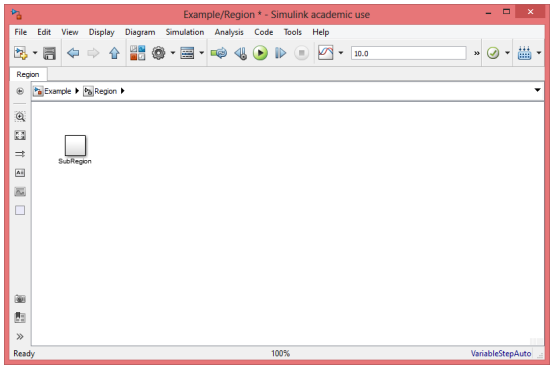

Figure 16: The subsystem "Region", which contains the 'SubGeographicalRegion' "SubRegion".

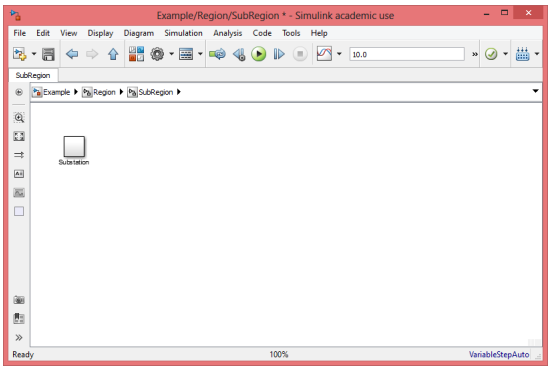

Figure 17: The subsystem "SubRegion", which contains the 'Substation' "Substation".

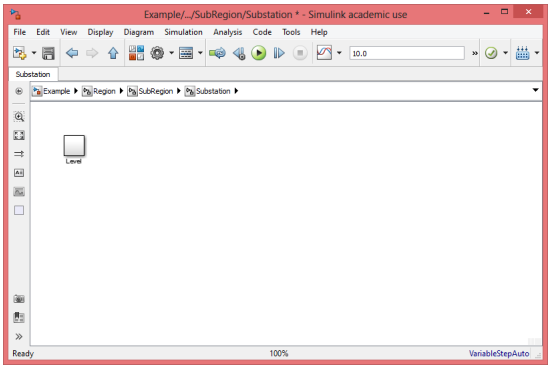

Figure 18: The subsystem "Substation", which contains the 'VoltageLevel' "Level".

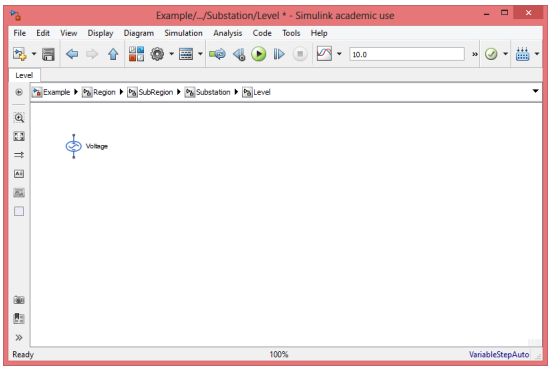

Figure 19: The subsystem "Level", which contains the 'BaseVoltage' "Voltage".

6. Conclusion

In conclusion, the files '**generatePowerSystem.m**' and '**parseCIM.m**' are **finished**. Only '**createSystem.m**' is **not finished**, yet, because there are **too many CIM classes** to implement. I decided to focus on just a few classes, which should be implemented first for a prototype. Even this amount of classes was too big. Only **six classes are fully implemented** by mapping the CIM classes to objects in Simulink. At this state, **simulations are not possible**.

What remains to be done is, to map further CIM classes to Simulink objects. Once, enough classes are mapped, the single objects have to be connected. After that is done, simulations should be possible.

Further development of the script could be done by implementing **more CIM classes**, **improve performance** - actually it is written to work well, not fast - or implement **further functions** like creating a CIM file out of a Simulink model.

1. REFERENCES

[1] The website of OpenGridMap: http://opengridmap.com/.

[2] The homepage of the International Electrotechnical Commission: http://www.iec.ch/.

[3] The Carbon Credits Market Model: http://trac-car.com/Carbon Credits Model HTML/index.htm.

[4] W3C's XML 1.0 Specification: http://www.w3.org/TR/REC-xml/.

[5] W3C's RDF Specification: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[6] The homepage of MathWorks: http://de.mathworks.com/.

[7] Wikipedia: Class diagram. https://en.wikipedia.org/wiki/Class_diagram.

[8] Wikipedia: Common information model (electricity). https://en.wikipedia.org/wiki/Common_Information_Model_(electricity).

[9] Wikipedia: Matlab. https://en.wikipedia.org/wiki/MATLAB.

[10] Wikipedia: Object diagram. https://en.wikipedia.org/wiki/Object_diagram.

[11] Wikipedia: Resource description framework. https://en.wikipedia.org/wiki/Resource_Description_Framework.

[12] Wikipedia: Simulink. https://en.wikipedia.org/wiki/Simulink.

[13] Wikipedia: Unified modeling language. https://en.wikipedia.org/wiki/Unified_Modeling_Language.

[14] Wikipedia: Uniform resource identifier. https://en.wikipedia.org/wiki/Uniform_Resource_Identifier.

[15] Wikipedia: Xml. https://en.wikipedia.org/wiki/XML.

[16] David Sardari Jose Rivera, Christoph Goebel and Hands-Arno Jacobsen, editors. *OpenGridMap: An Open Platform for Inferring Power Grids with Crowdsourced Data.* 6th Conference on Energy Informatics (EI2015), Karlsruhe, Germany, Christoph Goebel, 2015.