Rapid development of GUI Editor for Power grid CIM models

Sasa Devic[1], Lajos Martinovic[2], Branislav Atlagic[3], Zvonko Gorecan[4] and Dragan Tomic[5]

Abstract – **This paper presents a solution for designing a GUI editor for power grid CIM models, and a code generator that will ease the work on developing such editor and rapidly decreases time needed for development. The work contains basic description of CIM models and exchange procedure between clients participating in power trading process. Developed code generator relies on UML schema contained in a file created by a designer tool. The approach described here aims power grid CIM models, but it can be used for other CIM models as well. The general instructions for developing code generator are given. At the end, the overall CIM data handling process and resulting solution are presented.**

Keywords – **CIM model, GUI Editor, code generating, interoperability, ICEST 2012.**

I. INTRODUCTION

With constant growth of energy consumption and development of electrical power systems the need for connecting separate (national) transmission networks into one synchronous connection was recognized. In order to increase reliability and security, in 1951 *Union for the Co-ordination of Transmission of Electricity* (UCTE) was formed, which introduced UCTE standard for exchanging electrical network models between different TSOs (*Transmission System Operator*) operators. In order to increase reliability and security, in 1951 *Union for the Co-ordination of Transmission of Electricity* (UCTE) was formed, which introduced UCTE standard for exchanging electrical network models between different EMS (*Energy Managment System*) operators.[1]

But since then, many things have changed. The need to model data more precise, and to cover electrical elements not included in UCTE model, such as shunts, generators, transformer windings, switchers and so on, a new CIM (*Common Information Model*) model was developed. In 2009 UCTE became part of ENTSO-E (*European Network of Transmission System Operators for Electricity*). ENTSO-E accepted CIM standard as preferred, and in 2009 first interoperability (*IOP*) tests were made, although CIM model is still in developing phase.

IOP tests are held yearly, during the second week of July in Brussels, Belgium. Main purposes of IOP tests are joint testing of developing CIM model, comparison of different software solutions, promotion and wider application of CIM as standard. This paper represents one important part of a product that took part in IOP 2011, where it was well noticed along side with products from companies like Siemens, DigSILENT, Cisco, CESI, GE Energy and 10 other companies. That was its first participation.

Here we will point out that this work is logical continuation of work presented in [4], and main concepts are derived from that project.

[1]Sasa Devic is with TelventDMS D.O.O, Narodnog Fronta 25A-B, 21000 Novi Sad, Serbia, E-mail: sasa.devic@telventdms.com.
[2]Lajos Martinovic is with TelventDMS D.O.O, Narodnog Fronta 25A-B, 21000 Novi Sad, Serbia,
E-mail: lajos.martinovic@telventdms.com.
[3]Branislav Atlagic is with TelventDMS D.O.O, Narodnog Fronta 25A-B, 21000 Novi Sad, Serbia,
E-mail: branislav.atlagic@telventdms.com.
[4]Zvonko Gorecan is with TelventDMS D.O.O, Narodnog Fronta 25A-B, 21000 Novi Sad, Serbia,
E-mail: zvonko.gorecan@telventdms.com.
[5]Dragan Tomic is with TelventDMS D.O.O, Narodnog Fronta 25A-B, 21000 Novi Sad, Serbia,
E-mail: dragan.tomic@telventdms.com.

II. CIM MODEL

A. Basics

CIM is a set of standards for system integration and information exchange based on a common information model. CIM is maintained by IEC *(The International Electrotechnical Commission)*. The part of CIM standards accepted by ENTSO-E is designed for energy market systems. With codename iec61970, development is run by IEC TC57 WG13 (Technical Committee 57, Work Group 13). Profile referred to in this work is for transmission networks – CPSM *(Common Power Systems Model)*. At the time of writing active version of the standard is 15.31. Hereinafter, when referring to CIM-iec61970, profile CPSM, version 15.31, we will say only CIM.

The purpose of CIM standard is to define how members of ENTSO-E, using software from different vendors, will exchange network models as required by the ENTSO-E business activities. The following basic operations are sufficient for TSOs to satisfy ENTSO-E network analysis requirements: ***export*** (export internal network model so it can be unambiguously combined with other TSOs internal models to make up complete model), ***import*** (to import exported models from other TSOs and combine it to make complete model), ***exchange*** (every sent model must carry the data who formed it, which data brings and for which use case is designed for).

B. File structure

ENTSO-E CIM models are packed and exchanged as XML (*Extensible Markup Language*) data model. Data division among files is based on the kind of information in each file. This division typically divides logical groups and less rapidly changing information from those changing more frequently. Therefore, model information exchange is divided into eight files: *Equipment*, *Equipment Boundary*, *Topology*, *Topology Boundary*, *State Variables*, *Dynamics*, *Diagrams* and

Geographical data. Information from all six files can be combined into one "complete model", which joins all data. Changes on the model are exchanged by difference files that only contain information what are changed, new and deleted elements. When difference files are received, changes it carries are applied to the model.[2]

The CIM model itself is designed with *abstract* and *concrete* classes. Through those classes it maps physics of electrical power system, its states at the specific time, to the model. Abstract classes are used to ease the complexity of the system, they group and define base attributes and associations, dividing more and less general parts of the system. In contrast real (concrete) parts of the system are left to be described by concrete classes, which inherit much of its attributes and associations from abstract classes. Concrete classes are dependent on abstract classes. Still, there are concrete classes that do not inherit any abstract class. Data exchange involves only concrete classes. As an example *Voltage Level* class will be presented (see Fig. 1).

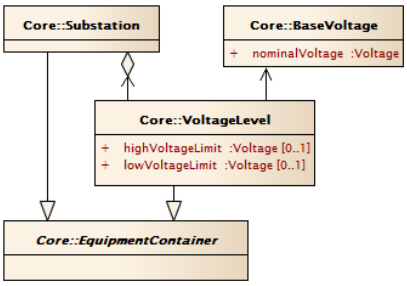

Fig. 1. Voltage Level (object-oriented model)

Voltage Level class inherits *Equipment Container*, as well as Substation class does. But, there are some attributes and associations in *Voltage Level* that do not exist in *Substation*, and vice versa. *Voltage Level* also has an association to *Base Voltage* and an aggregation to *Substation*. However, *Substation* doesn't have those connections.

This small part of CIM model is chosen to show the complexity of the model itself. Other elements of CIM model have more complicated associations and aggregations which would be hard to follow.

III. GUI EDITOR DESIGN

In this section we will try to analyze possible GUI design from designer point of view, with respect to existing solutions for this problem. As we know, a good GUI interface is an intuitive one; it reduces learning experience to plain visual search for commands that resemble verbs of spoken language. Work area should be well organized. We must keep in mind that interface is the one that attracts customers into buying a product, and background solution can only keep them.

Most solutions described in [3], that allow editing of all classes and fields in CIM model, tend to have display interface separated into two vertical sections, the left one – *navigation* panel, and the right one – *details* panel. Navigation panel usually consists of tree view, where root branches are elements that reference to no other element, their sub branches are elements referenced to them, and leafs are elements to which no other element references to. So, following those rules, elements that reference to multiple elements can be shown in tree view by their multiple representations (more than one occurrence). This can cause big confusion with users, especially considering the fact that elements are represented by their IDs, which mostly have no mnemonic meaning. We will also mention that to some elements many elements reference to, and that can recursively continue resulting in a tree with large hierarchical depth, which practically means – a lot of mouse clicking go get to the element that user needs. Search tool is usually added as additional tab in navigation panel and increases information density on one place. Details panel shows attribute details for one selected elements in navigation panels, where attribute name, type name and value have separate columns. Details panel is wider then navigation panel, and therefore it takes central place of application interface. But, unlike navigation panel, details panel offers much less information to the user.

We should use navigation panel to show with what types of elements we can work with, and actual list of elements move to the center of user's attention – to details panel. Attribute values of selected element from the list should be displayed right under the list, while attribute type should be shown only as a tool-tip. Navigation panel could be arranged as a tree that resembles hierarchy of abstract and concrete classes. By doing so, we will break the complexity of CIM model into logical groups. With this rearrangement we are able to set users focus to the center of our application, we have achieved a certain balance within the application commands, without forcing the user to specially learn how to use the new interface. Elements are organized into their logical groups, so looking for what user needs should not be a problem. A plan for this interface is presented in Fig. 2.

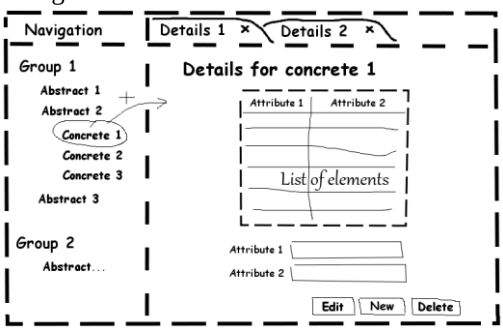

Fig. 2: Interface plan

IV. CODE GENERATING

Now, we have determined what kind of data is exchanged between different software vendors (CIM XML files) and we know in what form to present data in our GUI interface. The piece of the puzzle still missing in our software solution is *how*? We need to develop GUI that implements the idea that we want to realize.

This may sound like a simple task, but if we have to solve many simple tasks over large amount of different types of data, our *simple* problem becomes a *big* problem. We must keep in mind that for every field in every class comes around 100 lines of code, and for every class comes around 2.000 lines of code to correctly manipulate with classes and their

fields, to add, update and delete them all with respect to one another. Knowing that CIM, in version 15, has grown to a number of 337 classes, of which 222 are concrete and require a graphical representation, we get the picture how large this problem really is. If we consider the fact that for IOP 2011, CIM model was modified almost till the start of tests, our problem is even bigger. Since requests for changes on the model increases during last two months before the IOP, improper development plan could result in unwanted results. So, as presented in [4]:

The possible solution is to make a new problem, which will solve our first problem – to write a program that will write a program that we need.

Situation now is significantly changed. Even though database is no longer the center of our application, still it is possible to write a code generator that will write a program that we need. CIM standard for power grid has been developing using UML description language, which are shared though a common file type. This is the key, if we can programmatically read the file containing CIM diagrams, we can build a code generator that will write a code that we need!

In this work we will first generate the code, do some additional changes (if needed), and then compile it with the rest of application written by hand. Before going into details of code generation, first let us analyze the existing application where product of our code generator needs to fit in, technology in which application is build and possible technologies for developing code generator.

A. Existing application

Existing application is a desktop client for a WCF web service, written in C# language, in .Net framework, with use of WPF graphical subsystem. CIM models are read and sent from server side to the client side though a web service. We will concentrate our attention only on client.

Since client side is written in WPF, we will try to explain its main principles. WPF is presentation system for building Windows applications with visually stunning user experiences. Good advantage of WPF is the ability to develop an application using both *markup* and *code-behind* specifications. Markup, here called *Extensible Application Markup Language (XAML)*, is used to implement the appearance of an application; while using managed programming languages (code-behind) is to implement behavior, in our case C#. In general this is done to separate design from behavior, to allow designers to take a larger part in development process [5]. Therefore, we can conclude that code generator has to support generating both code-behind (C#) files and markup (XAML) files. Let us examine possible technologies for development of such code generator.

B. Possible technologies

For generating code-behind choosing possible technology was very straight forward. We can write required library by ourselves, like in [4], or we can use CodeDOM [6], a build in .Net API for programmatically constructing and compiling C# source code . General approach is to first construct source code, save it to a file, use built-in compiler to compile it and run it. Here, source code will be construct and saved into a file, and later compiled with the rest of developing application. This is recommended and fully satisfactory technology for needs in this project.

For generating markup, XAML code, things are not so clear, even though XAML is actually one more extension of XML. All built-in elements that appear in XAML can be specified though XML. Creating classes that resemble WPF classes and serializing them to XAML file could really ease the work for us. But, if we want to design reusable components that can be used just like system controls, we should create *user controls*. User controls can contain other (user) controls, resources, and animations, just like a WPF application. For each element from CIM model we need to create a few user controls that have references to other user controls, like specified in CIM model.

User control has root XML element with the same name – *UserControl*, where XML attribute *Class* specifies code-behind class. If user control needs to reference to other user controls, it needs to have an XML element that represents that user control, with the name from its *Class* attribute. This is the real problem with this approach.

In general we would write new user control by simple extending base user control class, and adding references to other already written user controls. But while code generator is running, we can't declare, initialize and reference to the new class from some other also new class, whose names we just read from UML diagram. We could try with constructing a class for serialization with needed references to some default user-control-reference class, and override its name before serialization to the name of a class it actually represents. But this also won't fit our needs, because we always have more than one reference to other classes. Overriding one element type will override its name wherever it appears. We will end up with always referencing to the last element that was overrided.

One more possible solution is to try inventing a new approach, to write our own, new library that will help us in generating wanted XAML files...

C. XAML Generator

By analyzing XML structure, which is also XAML structure, and also WPF structure in whole, we can notice that it reflects composite design pattern. If we *copy* WPF design pattern by using plain classes that have identical hierarchy and attributes, with simple overriding *ToString* method and saving the output to a file, we can get the result we need. Each class will print XAML code that corresponds to WPF class it represents. Printing all attribute values that differ from their default values, will also be included. Of course, like in WPF, all attributes are inherited from base to derived classes. Plain class that represent user control will have additional method called *GenerateReferenceCall*, for simple solution of problem we had with setting references in standard XML approach, explained in previous subsection.

The beauty of this solution is its simplicity and ease of use. Library that implements ideas here presented will allow us to

generate code for our application like if we wrote it by hand. That library we will call *XAML Generator*.

Since code-behind and markup are basically inseparable in WPF, to join the two libraries, CodeDOM and XAML Generator, we will generate both simultaneously. Functions that handle user events are defined in code-behind and called from markup. In that way view and controls are linked. In listing 1 we can see an example class from XAML Generator.

```
public class XButton : XButtonBase{
  public XButton(){}
  public override string ToString(){
    // print XAML element
    String atts = GetAttributeValues();
    String startElement = "<Button"+atts+">\n";
    String body = generateContent();
    String endElement = "</Button>\n";
    return startElement + body + endElement;  }
}
```

Listing 1: Example class from XAML Generator

The possible drawback is the fact that WPF is rather large subsystem of .NET, and writing generator to allow us full support for it would be very demanding. But this is not a real problem, at first, we will have support to generate only a small part of WPF, and gradually, as we progress in building our application, we will expand XAML Generator to fit our needs.

D. Implementation

After defining what we want user to see, navigation and details view, how to develop our code generator, CodeDOM and XAML Generator, we have left *easy* part to do – to generate code architecture to suite our needs.

By using UMLReader component, we will read the meta-data from the UML file, and use it in GUI code generator. This component was already developed as part of another project, and it won't be analyzed here. We will only mention that, like in [4], meat-data is stored into class model from which is easier to read classes names, attributes and relations among them. When meta-data is read into the memory, that will be meta-meta-data.

Navigator component we have already discussed in section III. Here we will only mention that this component is realized by knowing all the classes in CIM model, their names and their base classes.

Details View components will be separated into tree logical sections: tables, selectors and views components.

Tables are used to display a list of one type of CIM elements, where their attributes are shown in columns. Those are the simplest components. Search by criteria is also available for every attribute of string type.

Selectors relay on tables. They are used to select one element from a list of elements of certain type. This will be very useful for setting associations and aggregations among different CIM elements.

Views relay on both tables and selectors. They allow creating new element for the model, modifying and deleting existing elements selected in tables list. For setting associations and aggregations, selector components are used. View components are the most complex of all, because they take care of new, modified and deleted elements, storing them into three separated list for sending to the server side.

This code generator is designed to cover some of specific needs of ENTSO-E CIM model, but it can be applied for other CIM and UML models as well. Appearance of produced CIM GUI editor is given in Fig. 3. Sum of all generated lines of code, for 220 concrete tables, is 355,997 lines.

Development process is presented in Fig. 4.

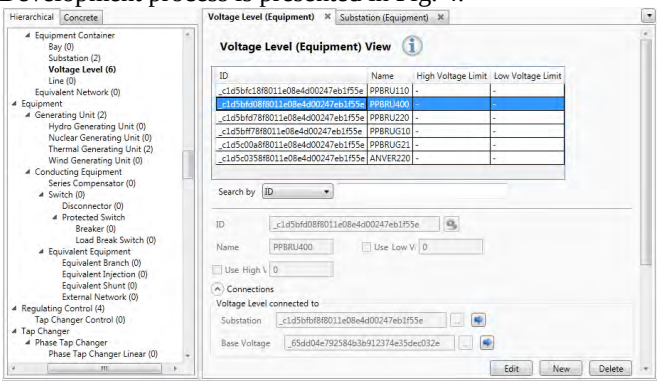

Fig. 3: Produced CIM GUI Editor

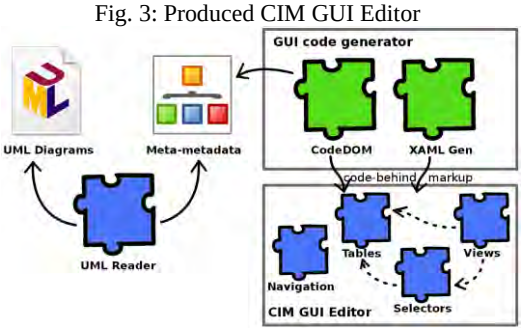

Fig. 4: Development process

V. CONCLUSION

In this work we have explained what CIM models are, and how they are exchanged. We designed more user friendly graphical interface. Code generator is described, for both code-behind and markup specifications. Solving problems with code generators allows us to develop applications much faster, with fewer errors and with less people on the project. The most important advantage of code generating is that it allows us to easily adopt our application on future (frequent) change requests, which is becoming a major obstacle for any large application.

REFERENCES

[1] D. Kirschen, G. Štrbac "Fundamentals of Power System Economics", 2005.
[2] ENTSO-E, "UCTE CIM Model Exchange", component interface exchange, revision 1.0, version 14, 2009.
[3] ENTSO-E, "Interoperability test: CIM for System Development and Operations – Final Report", 2010.
[4] S. Devic, B. Atlagic, Z. Gorecan, "Database modelling and development of code generator for handling power grid CIM models", ICEST, 2011.
[5] Ted Hu, "WPF Series Getting Started", Microsoft MSDN, 2010.
[6] Luke Hoban, "C# 3.0 and CodeDOM", Microsoft MSDN, 2007.