Database Modelling and Development of Code Generator for Handling Power Grid CIM Models

Sasa Devic[1], Branislav Atlagic[2] and Zvonko Gorecan[3]

***Abstract* – This paper presents a solution for modelling database, in order to store and manipulate CIM models, and code generator that will ease the work on developing support system for that database. The work contains basic description of CIM models and exchange procedure between clients participating in power trading process. Developed code generator relies on database relational schema. It is designed for power grid CIM models, but it can be used for other models as well. The general instructions for developing code generator are given. At the end, the overall CIM data handling process and resulting performances are presented.**

***Keywords* – CIM model, database, code generating, ICEST 2011.**

I. INTRODUCTION

Since general use of electrical energy has started, till the end of the XX century, the customers were forced to buy energy from monopolistic power supply companies which covered the wider area they lived in. Customers had no choice to choose between different, probably more affordable, companies. Economic experts rightly claimed that monopolistic companies are starting to slow down the technological progress. If more companies were fighting for the costumers, the final result would brought a better and cheaper energy supplies to the customers.

Events that followed were expected. Big, monopolistic, usually state companies, were reconstructed into large number of smaller companies or sectors that took part in managing different parts of once one company. For this market to operate, it was necessary to introduce auction houses for electrical energy trading. Auction houses work as a agency agent between different participators in trade, and therefore they need specialized software tools for trade. Opposite to other markets of energy resources, in trading with electrical energy the key role has the knowledge of physics of the system or, in other words, the knowledge of electrical supply network and its capabilities. For those needs a unique model for describing every network of different vendors was developed[1].

For developing software tools that would operate with such model, specialized software companies started to form. Those companies required knowledge of both electric power engineers and software engineers to develop a unique solution for the problem. This work gives contribution to those efforts.

[1]Sasa Devic is with Telvent DMS D.O.O., Sremska 4, 21000 Novi Sad, Serbia. E-mail: sasa.devic@dmsgroup.rs.

[2]Branislav Atlagic is with Telvent DMS D.O.O., Sremska 4, 21000 Novi Sad, Serbia. E-mail: branislav.atlagic@telventdms.com.

[3]Zvonko Gorecan is with Telvent DMS D.O.O., Sremska 4, 21000 Novi Sad, Serbia. E-mail: zvonko.gorecan@dmsgroup.rs.

Since 1951, UCTE (*Union for the Co-ordination of Transmission of Electricity*) standard was used to exchange electrical network models between different EMS (*Energy Managment System*) operators. 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 (*European Network of Transmission System Operators*). ENTSO accepted CIM standard as preferred, and in 2009 first interoperability tests were made, although CIM model is still in developing phase[2].

II. CIM MODEL

A. Basics

The CIM Model Exchange Profile is a ENTSO standard that is based on the CIM standards produced by IEC WG13 (*The International Electrotechnical Commission, Work Group 13*). The purpose of CIM standard is to define how members of ENTSO, using software from different vendors, will exchange network models as required by the ENTSO business activities. The following basic operations are sufficient for TSOs (*Transmission System Operator*) to satisfy ENTSO network analysis requirements: ***export*** (TSO may use the profile to export its internal network model in such way that it can be easily and unambiguously combined with other TSO internal models to make up complete models for analytical purposes), ***import*** (TSO must be able to import exported models from other TSOs and combine it to make complete model), ***exchange*** (any model, covering any territory, sent to any other party, must carry the data who formed it, which data brings and for which use case is designed for).[3].

B. File structure

ENTSO CIM model 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 less rapidly changing information from more rapidly changing information, setting up the situation where some exchanges are smaller because they only contain files that have changed. Therefore, model information exchange is divided into three files, TSO equipment model, TSO topology and State variables. Information from all three files can be combined into one "complete model", which concatenates all data.

The CIM model itself is designed with *abstract* and *concrete* classes. Through those classes the physics of

electrical power system, its states at the specific time, are mapped 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. Again, 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. Anyway, data exchange involves only concrete classes. As an example *Tap Changer* class will be presented (see Fig. 1).

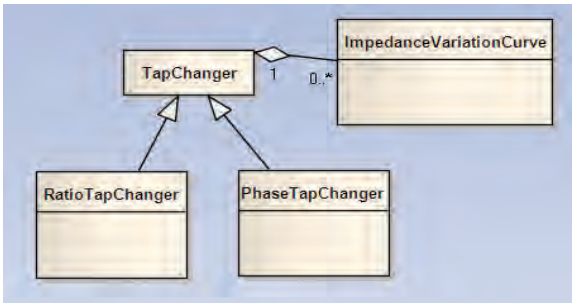

Fig. 1. Tap Changer (object-oriented model)

Tap Changer class is inherited by *Ratio Tap Changer* and *Phase Tap Changer*. Both, *Ratio* and *Phase Tap Changer* have all the attributes and associations of the *Tap Changer*, but with addition of a few of its own. There are some attributes and associations in *Ratio Tap Changer* that do not exist in *Phase Tap Changer*, and vice versa. On the other hand, if some concrete class, like *Impedance Variation Curve*, has an association to *Tap Changer*, which is an abstract class and can not exits in data exchange, it actually associates to either *Ratio* or *Phase Tap Changer*. Association to the abstract class is a true problem for mapping object-oriented model (hierarchical model) to the database relational data model (flat model), which will be more explained in the next chapter.

Tap Changer is chosen because it is simple enough for an example and it has all kinds of associations that we wish to demonstrate. Other elements of CIM model have more complicated associations which would be hard to follow, but with the same types of associations like *Tap Changers* has.

III. Database Modelling

As we know, basic elements of which relational databases consists are tables (which are composed of one or more columns) and relations between them. The relational data model has been around for many years and has a proven track record of providing high performance and flexibility. But, there is no possibilities of defining real inheritance among data tables or defining an abstract table. Databases are designed to contain large amounts of data, and to allow quick and relatively easy way to access them. Any kind or mapping object-oriented class model to relational database model is not completely possible, modeling of the two do not follow the same design. To solve this problem, we have implemented some patterns:

- For each abstract class that we have in our object-oriented model, appropriate table must be created (in this work we will call them "abstract tables", like in [4]).
- For each concrete class, appropriate table must be created (in this work we will call them "concrete tables", like in [4]).
- For each association to other class, abstract or concrete, appropriate relation to the table that represents that class must be made.
- Inheritance is mapped as identification relation to the table that represents "*parent*" class (key columns of a "*child*" are all from "*parent*" table). The exception from this rule are abstract and concrete classes that have no "*parent*" class.

So far this is straightforward, but with large amount of redundant data! If we store all data that abstract classes define, very quickly our database is going to grow enormously. In order to prevent that from happening, since all data defined in abstract classes must be withheld within their inheritors, can we remove all tables that represent abstract classes and use only tables that represent concrete classes? By doing that, relational model would simplify and reduce amount of data in database, but there is a problem with that...

If there is a class with an association to the abstract class, in relational data model we cannot model that as a relation to the tables that represent concrete classes that inherit that abstract class. In relational model we cannot map one object-oriented association to a multiple database-oriented relations, depending on the number of how many classes inherit aimed abstract class. For example, in table *Impedance Variation Curve* we would have to have relations to both tables that represent *Ratio* and *Phase Tap Changer*, since they both inherit *Tap Changer* class in object-oriented model. If we could define a reference with a choice clause, that could be one of possible solutions for the problem (but that is not possible). We are forced to retain tables that represent abstract classes. The solution to the problem is to use abstract tables only as key containers, to which we can set relation through which we can find needed concrete table. All data is stored in the concrete tables only. Here we will mention that we need abstract tables from maintenance reasons as well. CIM model is still in development process and it is expected to change. Relational model for *Tap Changer* is shown in Fig. 2.

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 store such data in our database. The piece of the puzzle still missing in our software solution is *how*? We need to have a support to insert the data into database and later to read from it and manipulate with data.

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. Developed relational model has 86 tables with very complex relations among them. 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.

We have designed our database relational model to reflect object-oriented model, we can use its meta-data (data that

describes data) to generate program code and use it to manipulate with database. Code can be generated in such way that will allow us to manipulate with objects that reflect their states back to the database.

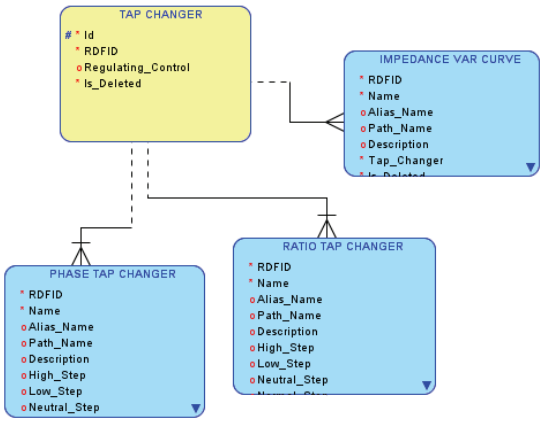

Fig. 2. Tap Changer (relational model)

A. Database reader

First step is to develop *Database reader*, a software component that reads meta-data from database relational schema. All database meta-data is possible to read. For every table that we have created, we can read its name an other meta-data related to table. When we know what tables are present in database, in a similar way we can read their columns meta-data. For each column we can read its name, data type, size, numeric precision and scale, is mandatory, is read only, and most importantly is it a primary key column, a foreign key column, or both. We can even read column constraints on data it can contain. Important to mention is that based on foreign keys (imported keys), we can find out which primary keys from one table are used as foreign keys in other tables (exported keys).

To ease our search through meta-data, we have created an object model where we will store meta-data that we have read. Since it's storing a description of meta-data, which is data that describes data, that model is actually a meta-meta-data model. All other parts of our code generator rely on this meta-meta-data model.

B. Database procedure generator

In order to fully exploit database we have to use database procedures. Insert, update and delete statements are very slow when compared to equivalent procedures. The main drawback when using statements is that they need to be parsed first, and then executed. In contrast, procedures executes immediately.

To avoid writing all required procedures manually, we have created a *Database procedure generator*. Its job is to generate (write) those procedures, using meta-meta-data. We know the names of tables, their columns and data types.

For insert procedure we need to define input parameters, which are values for columns of a table in which we wish to insert a new row. Next step is to define what our procedure needs to do, in this case to insert a new row into the table. Table name, its columns, and type are used as input parameters in this process.

For update and delete procedures we need to do similar work, except we are not generating insert, but update and delete statements. Needed syntax is different, but the logic is similar for all.

Views are also generated by this component, to allow us faster and easier read of data.

All in all, with some additional procedures that will be explained later in this work, we have generated almost 15.000 lines of code for those procedures, and 2.500 lines to drop them later.

C. Database API generator

To work with database in a way like we work with objects in our program language, we need to develop a communication layer that will allow us to do that. *Database API generator*, based on meta-meta-data model, is developed to generate an object-oriented model who's states will be reflected on database.

We need to find a way to generate a component layer to stand between database and the rest of our application. Lets observe our tables in database. Each table is consisted of rows and columns. Columns describe a certain data of one type, with specific constraints. We will map columns like attributes of a certain class. Rows can be viewed as sections of data of a certain table. We will map rows like a classes, where class name is delivered from table name. Additionally, based on imported keys we have generated association to the class that represents a certain table in database, and based on exported keys – collection of associations. By generating classes that we need, we are allowed to use them in our code like any other class that we have wrote by hand. The example of this code is given in Listing 1.

```
class TapChangerEl {
    // Exported Relations
    List<ImpVarCurveEl> impVarCurveEls;
}
class ImpVarCurveEl : CurveEl {
    // Imported Relation
    TapChangerEl tapChangerEl;
}
class RatioTapChangerEl : TapChangerEl { ... }
```

Listing 1: Generated code example

Classes that we have created must have support to call database procedures that we created by *Database Procedure generator*. By applying similar logic we are able to generate class functions that are able to prepare the call to database, set needed parameters, execute the call, and finally do additional work with result. Class functions generated for that are insert, update and delete.

Database API generator is also in charge for generating functions for fetching data from database and creating class objects from it. Fetch functions include various forms, depending by which criteria data is read (by primary keys, by foreign, by columns, by no criteria or by combination of

them). Fetch functions rely on views generated by *Database Procedure generator*.

This code generator is designed to cover some of specific needs of CIM model, but it can be used in general use, as well. Some of specific needs are presented in next section. Sum of generated lines of code, for 86 tables, is 506,781 lines.

V. Import-export Model

So far, our model, that we got as the output of code generator, is covering all functionality we need, with ability to be easily expanded in the future. But, in order to get any data from concrete tables, first we have to fetch their abstract tables that index them! Number of calls to database can be multiplied several times depending on the number of abstract tables through which we have to search to get the data in some concrete table. In general, as deeper the hierarchy of object-oriented model is, as slower the communication to database will be.

As we mentioned before, data contained in abstract tables is also contained in tables that "inherit" them (basically only keys). Since the aim is to reduce number of calls to the database, we could read only concrete tables, and by doing that we get data from abstract tables as well. This approach reduces number of calls to the minimum, and there is no bottleneck to the database. To compare performances without and with this principle see table I (old and new for export).

Based on that idea, to read only concrete tables, it is possible do the same in opposite direction. Database procedures will take parameters for inserting a row into a concrete table, first appropriate data will be inserted to all needed abstract tables and after that to the aimed concrete table. By applying this principle we have achieved performance acceleration in both ways. For this we had to expand both *Database Procedure generator*, to create such procedures in database, and *Database API generator*, to give us access to those procedures. To compare performances without and with this principle see table I (old and new for import).

TABLE I
PERFORMANCE TABLE FOR IMPORT AND EXPORT CIM MODEL WITH 50.000 ELEMENTS (54MB FILE)

	Import [s]		Eksport [s]	
No.	*old*	*new*	*old*	*new*
1	1173,0	188,9	49,42	4,30
2.	1377,7	209,5	47,42	2,69
3.	1014,6	212,8	47,58	2,71
4.	1470,2	183,6	47,79	2,66
5.	944,5	148,3	48,35	2,64

VI. Software Package

Software application is build with two basic components, *Cim Manager Lib*, *Database Api* and *Cim Mediator*. Application development process, that we are about to explain, is presented on Fig. 3.

By the standard, CIM model is exchanged with XML files. Since C# programming language and .NET have well developed technologies to support manipulation with XML, software application is developed using those technologies. This covers *Cim Manager Lib*, as described in [5].

To communicate with database *Database Api* is generated. It is a wrapper component to ease the access to the database. Database supported by this project is Oracle 10 XE, an library used for communication is ODP.NET (*Oracle Data Provider for .NET*). ODP.NET. It is possible to add support for other databases as well.

Cim Mediator is a component for joining the *Cim Manager Lib* and *Database API* into one solution. It represents the entrance point of application and contains and exposes functions for import, export and for fetching functions of entire CIM model.

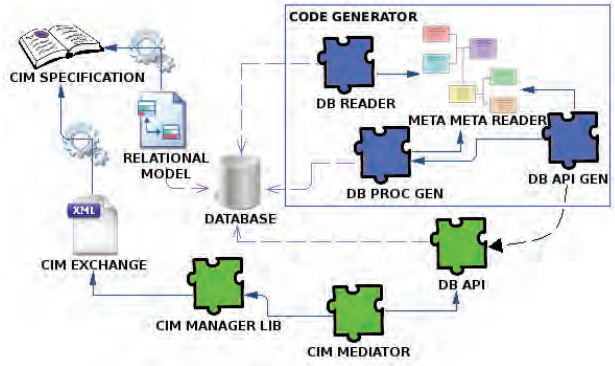

Fig. 3: Development process

VII. Conclusion

In this work we have explained what CIM models are, and the way how to work with them when stored in database. We have presented components that help us work with database. Code generator is described, through which we solved the problem of writing large amount of code in limited time. The most important advantage of code generating is that allows us to relatively easy expand our generator depending on current needs. Another advantage is that it gives us the power to go even further with code generating. If we need to have client-server architecture with our database, we can make an extension for code generator and generate web service and client that we need. This could be done as an update in the future.

References

[1] Daniel Kirschen, Goran Štrbac „Fundamentals of Power System Economics", 2005.

[2] Union for the Co-ordination of Transmission of Electricity CIM interoperability Test – *EPRI*, 2009.

[3] Dragan Tomić, Ranka Slijepčević, Lajoš Martinović, Nemanja Živković, „Validation and merging of national transmission network models into one interconnective model", 2009.

[4] "UCTE CIM Model Exchange", component interface exchange, revision 1.0, version 14, 2009.

[5] Lajoš Martinović, Ranka Slijepčević, Nemanja Živković, "Software tool for conversion between power system models", 2010.