ITC 3/46
Journal of Information Technology and Control
Vol. 46 / No. 3 / 2017
pp. 319-332
DOI 10.5755/j01.itc.46.3.14340

Development of a Database for the Common Information Model of Power Grids

Received 2016/07/07 | Accepted after revision 2017/07/13

http://dx.doi.org/10.5755/j01.itc.46.3.14340

Development of a Database for the Common Information Model of Power Grids

Saša Dević

Schneider Electric DMS NS, Narodnog fronta 25A, 21000 Novi Sad, Serbia
e-mail: sasa.devic@schneider-electric-dms.com

Ivan Luković

University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
e-mail: ivan@uns.ac.rs

Corresponding author: sasa.devic@schneider-electric-dms.com

The ongoing development of a complex model for power grid networks, based on the Common Information Model (CIM), is dealing with design, operability and exchange of data among various power grid operators. This paper presents a methodological approach to development of a database that supports an easy storing and managing of active CIM instances, as well as their historical versions. To facilitate the implementation of the proposed approach, we apply a paradigm of automatic programming. Our code generator eases the work on developing an API communication layer over the database and allows faster response on CIM changes. Finally, we present a performance cost analysis on test models. By this, our intention is to contribute to a wider acceptance of CIM in power grid networks.

KEYWORDS: Common Information Model (CIM), database modeling, code generators, power grid, model to code transformation.

1. Introduction

Power systems have been increasingly used since the middle of the 20th century. At the beginning, power grids were simple, isolated systems. In order to monitor network states, to track power grid states, inputs

and outputs, Transmission System Operators (TSOs) had to have a model that would provide them with needed data. In 1951, the first European association appeared – the Union of the Co-operation of Transmission of Electricity (UCTE) [7]. Since the UCTE's appearance, electrical energy production, transportation and distribution became more separated between different companies.

After the separation of transmission, generation, distribution and trading activities of the electricity sector, a possibility of an easy exchange of operational data has become even more important. Electrical markets have triggered an increase of cross-border power flows between countries. This leads to significant variations in generation patterns, displacing substantial amounts of electricity from one area to another, or from one hour to another. Therefore, exchanges of data have increased dramatically during recent years [16].

Since UCTE has been established, its model for describing power grids, UCTE DEF, has been widely used. UCTE DEF is power flow oriented model. Since it carries only the basic, necessary, information about power grid nodes, lines and border nodes, we could say that it is relatively simple model. UCTE DEF has been designed to meet requirements of European TSOs in the middle of the 20th century. For a long time, it was recognized as a satisfactory model. However, its simplicity appeared to be the main cause of its serious limitations in recent decades.

To address the UCTE DEF limitations, a development of a new model for power grids was initiated in 1999. It is the Common Information Model (CIM). CIM is a still evolving model, and now maintained by the *International Electrotechnical Commission* (IEC) [11]. As part of its development, cooperation and coordination, *European Network of Transmission System Operators for Electricity* (ENTSO-E) [7], the successor of UCTE, also accepted CIM as the preferred model for describing power grids. CIM is a network oriented model that provides a common definition of management information for systems, networks, applications and services, and allows extensions [11].

Development of CIM is followed by a series of standards. Part of CIM for energy market systems is covered by the standard IEC 61970, run by the IEC Technical Committee 57, Work Group 13 (TC57 WG13). At the time of writing, its current version is 2.4.15, while the version of its model is 16.25. A profile referred in our research work is Common Grid Model Exchange Standard (CGMES) that is used for transmission networks. Hereinafter, when referring to CIM, we refer to the IEC 61970, profile CGMES, standard 2.4.15, and model version 16.25. The CIM model is defined via UML.

We may say that CIM defines a meta-model for power grids, as CIM instances are models of concrete power grids. Here, we observe CIM as the initial meta-model of our approach based on Model-Driven Development (MDD) paradigm [4], [5].

Despite the fact that CIM is a continuously evolving model, it has already been widely accepted and the need for its software support is constantly increasing. Vendors that use CIM in their applications are dealing not only with the problem of how to use it in their power functions, but also how to store CIM instances. In regard to this, it is not only the issue how to save initial model instances in a database, as a repository of CIM instances, but also how to track all their further changes, as there is a strong requirement to provide a possibility of restoring a model instance to some of its previous states.

From the previous experiences reported in [27], [10] and [28], it follows that for the development of a database repository for instances of complex models as CIM instances certainly are, there is no a straightforward solution. From this, in our research we assume that CIM can be implemented as a relational database schema, while CIM instances are stored in a relational database, in an optimized way, without losing any data.

The goal of our research is to propose a methodological approach with a generic database repository structure in support of efficient storing of CIM instances, tracking changes without redundant data, and their restoring to previous states. In support of our approach, we propose a software application developed for CIM in power grid.

In this paper, we propose a repository structure for storing both current and previous (past) states of CIM instances under a relational database management system (DBMS). Here we introduce the notions of an *active model* and *historical model*. The *active* model supports a specification of the current states of CIM instances, while *historical* model supports a specifi-

cation of the previous states of CIM instances. The development of active and historical models is organized in two phases of our methodological approach.

Our software application for CIM in power grid fully supports both, the active and historical models. It is designed in a way to meet expectations and logic of work of power engineers. As CIM is a constantly evolving model, an important requirement for this application is to be easily maintainable. To facilitate such a requirement, we have applied a paradigm of automatic programming in development of our application. Thus, we have developed our code generator that eases the work on developing an Application Programmatic Interface (API) communication layer over the database that allows faster response on CIM changes. By this, complexity and the amount of required work needed for writing code is reduced significantly.

The rest of the paper is organized as follows. Works related to our research are presented in Section 2. In Section 3, CIM is presented in more details, with its inner organization. Section 4 presents the first phase of our methodological approach – how to model the database to be accessed in a relatively easy way, without loss of speed. The second phase of our methodological approach is presented in Section 5. The section explains how to structure historical part of database to provide tracking changes made on model instances, and restore them to any selected state. In Sections 4 and 5, a code generator is introduced to facilitate the implementation of the system. Evaluation of the results, with time performance analysis is given in Section 6.

2. Related work

In the survey of related works, we identified the following groups of references: i) works about CIM standardization and the importance of formalizing CIM as a standard (standardization improves CIM interoperability among various software vendors); ii) works about CIM in a practice use in power grids (this group of references is of an interest in searching for concrete examples of potential applications of our work); iii) works about strategies of storing CIM instances in a database (these references impact the selection of a proper approach to data organization and also our approach of development of a database system for CIM); and iv) works about automatic programming paradigm and its influence on a reduction of development time, and increasing reliability of produced code. We present a selection of works that mainly affected the development of our code generator.

In [16] and [7], some important steps in moving from UCTE DEF to CIM are discussed. Ivanov and Chury in [16] argue that "the current version of the UCTE ASCII data exchange format (UCTE DEF) turned out to be insufficient for planning purposes because some data is missing and some network elements are not described in an appropriate level of detail." In addition, Britton and deVos in [7] conclude that successful implementation and the usage of the CIM will significantly improve the accuracy, quality and reliability of cross-TSOs data exchanges. In this research, the current version of CIM [11] is applied in a power grid. Moreover, Britton and deVos in [7] and Britton et al. in [6] propose ways of its development. Our approach relies on the data exchange process as it is proposed in [16].

Since it has been introduced by IEC, CIM has been extensively analyzed and used in power grids. In [20], CIM is used as the data model in the algorithm for finding the catalogue of topologies in a power system. In [8], it is a data model in algorithm for partitioning power grid networks. In [24], CIM is used in the scheduling algorithm for controlling power grid. Our approach is also based on CIM, as it is well known and recommended by international authority body. CIM is designed to be robust enough to support different granularity of data [15]. As described in [26], both bus-breaker and node-breaker models can be represented though CIM. The references [8], [20] and [26] give examples of a CIM usage in power engineering. Those are the typical examples of projects, where our approach will be applicable, as an extension that will provide a database support for storing and managing CIM instances.

One of the two papers of a narrow interest for our research presents a comparison of two strategies for storing CIM instances, relational and RDF-based database approach [28]. CIM itself does not offer any official database models, so the focus was to find pros and cons of using a relatively new RDF-based database versus well-established and ubiquitous relation-

al database, for storing CIM instances. As concluded in [28], the main drawback of relational database is in that all data have to be mapped to and from CIM-XML format. However, due to being faster, the relational database is more preferred. In [27], Ravikumar et al. propose a *CIM oriented database (CIMODB) design through the ORM*, similar as we use in our approach. Both Ravikumar et al. [27] and Schulte et al. [28], advocate a selection of a relational (SQL) database in spite the growing popularity of NoSQL, more precisely RDF solutions. A ubiquitous use of relational DBMS, as described in [9], precludes the use of other technologies such as NoSQL, especially for federated data schemas [27]. In many other works, as in [19] and [23], the authors are slightly reserved towards NoSQL databases because of the lack of standards, consistency, familiarity, maturity and maintenance. Primarily because of the results presented in [27] and [28], as well as in [9], [19], [10], and [23], a relational database approach is used in this work.

Automatic programming, as a programming paradigm is heavily used in many software development projects from the very beginnings of software programming [17], [21]. Since CIM is described with 600 classes, its specification belongs to a class of large and consequently complex models. The probability to make logical errors in designing such systems is high. The amount of time needed to develop database procedures and API communication layer by hand is also high. Therefore, we identify a need for the development of a code generator to support the process of implementation of CIM as a model under a DBMS. A code generator takes a high-level description as its input and generates lower level code [25]. That is, the input specification for generators is simpler and shorter than the generated code [14]. In [18], UML with sequence diagrams is used in order to produce application that would better reflect designed process. In [1], Ablonskis and Nemuraitė detect model-to-code transformations, which can later be reused for composing templates for generating a program code. The authors in [3], [12] and [2] start from a database model and templates to generate different layers of their applications. In addition, in [22], templates are used to transform model and generate Software-as-a-Service applications. In our research, we use a UML representation of CIM as an input specification of our code generator.

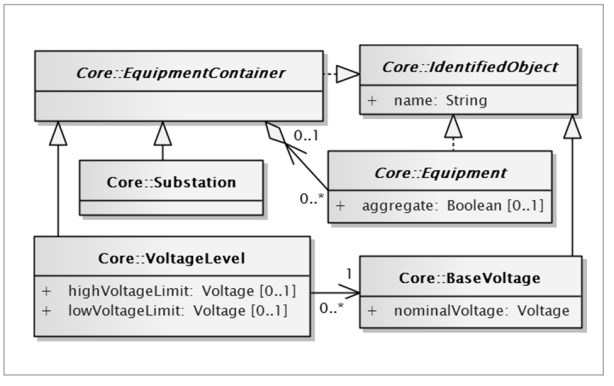
3. CIM structure and data exchange

One of the main purposes of CIM as a standard is to define how members of ENTSO-E, using software from different vendors, would exchange network model instances as required by the ENTSO-E business activities. Therefore, in this section we present the CIM model with its inner structure, as well as the process of data exchange based on it.

CIM instances are stored in XML file format. Data are divided into nine files: *Equipment, Equipment Boundary, Topology, Topology Boundary, State Variables, Dynamics, Diagrams, Geographical Data,* and *Steady State Hypothesis*. Information from all the nine files represent one *complete CIM instance*.

CIM is a hierarchical model that comprises *abstract* and *concrete* classes. Through those classes, CIM maps physics of electrical power system and its states at the specific time (every hour). Abstract classes are used to ease the complexity of the system; they group and define base attributes and associations, differentiating between more and less generic components of the system. In contrast, real (concrete) components of the system are described by concrete classes, which inherit much of the attributes and associations from the abstract classes. Concrete classes are dependent on abstract classes, as shown in Figure 1, which is an excerpt of the CIM model. In Figure 1, Equipment Container and Equipment represent abstract classes, while Substation, Voltage Level and Base Voltage represent concrete classes. Here, we are not discussing the meaning of those classes, as we are only interested in their relations.

Figure 1
CIM UML class diagram



Changes made on a model instance are exchanged by difference files that only contain information of what is updated, what is new and what is deleted. When difference files are received, they are applied on the model instance [11]. One of the main advantages of CIM is that data are maintained without the need to exchange the whole model instance [16], [7]. Changes done on one model instance, made by one company, are exported to a difference file and sent to other interesting parties who need to maintain their model instance of the same network. Difference files are XML formatted files.

In Section 4, we present a modeling process of the operational database, which relies on the CIM hierarchical structure. Besides, CIM exchange process based on the usage of difference files is a basis for developing methodological approach for historical model described in Section 5.

4. Active model

In this section, the first phase of our methodological approach is presented. The active model (Figure 2) is described, through which we model CIM oriented database aimed at storing CIM instances. The active model is a representation of the CIM model, where each class in CIM has its active model representation. As we select a relational data model paradigm for storing CIM instances, our active model is implemented under a relational DBMS. We call this database the *active* database.

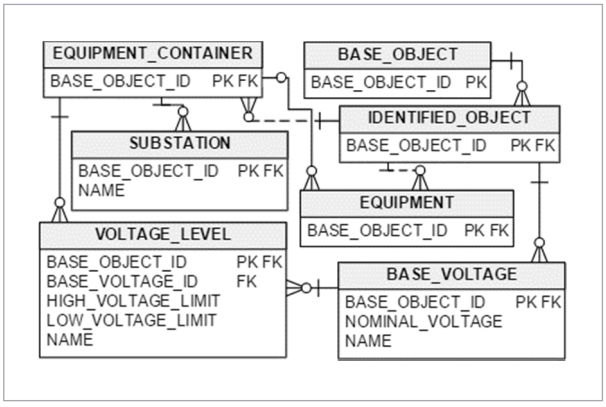
In this phase of our approach, the primary goal is to provide storing of the CIM instances, as well as easy and fast access to them. To achieve the goal, the phase is divided into three steps.

The first step is to perform an analysis of the active model in comparison to the CIM model. Next, we discuss two typical approaches for creating models, similar to our active model, and why they do not meet our goals. The second step is to formulate a procedure for creating our active model. The final, third step is to implement an active database. In the following text, we elaborate each of the steps, in more details.

In the first step, we describe the active model and how we create it. In Figure 2, an excerpt of the active model is shown. By the CIM terminology, abstract classes are mapped to so-called *abstract* tables, while concrete classes are mapped to *concrete* tables. In the active model, we introduce BASE_OBJECT table that represents Base Object class, which is not a part of CIM. We introduce it to ensure that all tables will have a surrogate primary key (attribute BASE_OBJECT_ID), which becomes a foreign key from the table that represents the Base Object class. Abstract tables that represent Identified Object, Equipment Container and Equipment classes are created without their attributes (columns) and references (foreign keys), as it is more convenient to have these attributes in their child, concrete tables that represent Substation, Voltage Level and Base Voltage classes. Therefore, these concrete tables include attributes from their *parent* tables.

Figure 2

A database schema of the active model



We discuss two typical approaches that could be used to create a model that is conceptually similar to our active model. Since CIM is specified in UML via Enterprise Architect, the same tool could generate SQL code for our active model. Next, since classes generated from CIM UML can be mapped to their table representatives, the same approach is also an option for our active model. As such, object-relational mapping (ORM) is used. Many frameworks could be used for ORM. However, in a model as complex as CIM is, with many hierarchical levels, a potential problem is that CIM has relatively large number of abstract entities [20]. If simply table-for-class is created, for reading data from any table it would be hard to write a simple select query. A join clause has to be used to reach each

of parent tables that contain data mapped from parent classes. The more join clauses we have, the slower the query will be [10]. However, our goal is to provide easy and fast access.

In the second step, we perform ORM in a specific way. Firstly, we map class attributes to their table representatives only for concrete classes, with the inclusion of inherited attributes. Secondly, tables representing abstract classes are created with keys only, without columns included. In our approach, abstract classes are needed to properly place the relations. As an example, in the model excerpt presented in Figure 2, Equipment requires a relation to Equipment Container. Equipment Container is an abstract class representation, as it generalizes either Substation or Voltage Level. Finally, parent references are mapped as foreign keys in concrete, child tables only. By doing so, it is possible to fetch all needed data from just one table, without using join clause. By this, all data attributes are pushed *down* to concrete tables, while abstract tables form the *skeleton* of the model.

In the final, third step, we present the implementation process of the first phase of our methodological approach. CIM model has almost 600 classes, around 200 of which are concrete classes. Here we map CIM-XML structured data into the relational database.

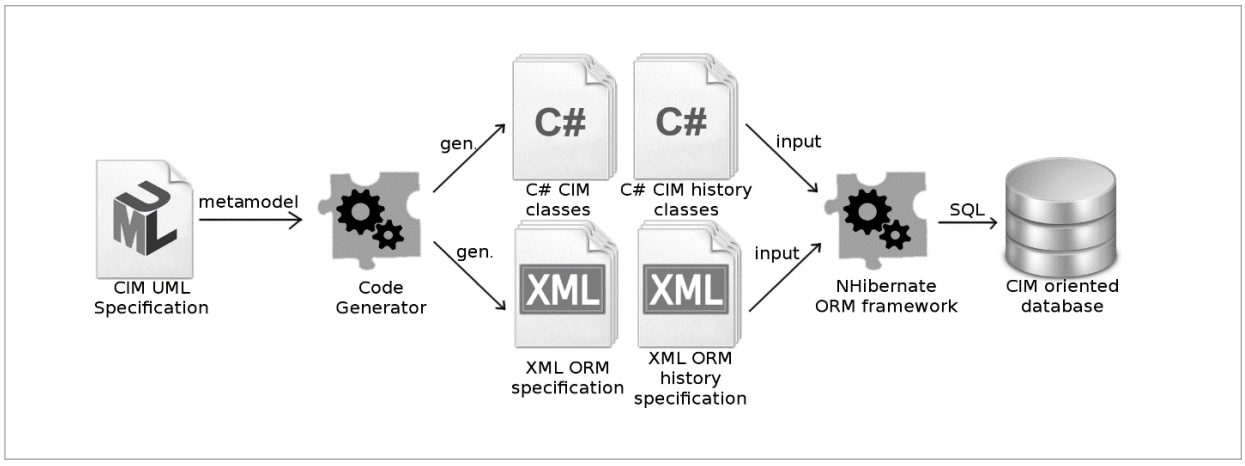
Implementation efforts for writing SQL commands for creating our database schema, database procedures and an API communication layer can be quite high. Therefore, we propose creating a Code Generator. A development process defined by our methodological approach is presented in Figure 3. Our Code Generator reads the CIM UML specification. From the CIM UML specification, we generate the CIM object model, comprising the C# CIM classes. CodeDOM framework has been used in developing the generator [13]. Next, the XML files are generated to specify ORM of the CIM object model. The XML ORM specification is created in accordance to the approach already presented in the previous step. Finally, having the object model and ORM specification, an active database is created by using the ORM framework NHibernate. The *history* elements, presented as the 'C# CIM history classes' and 'XML ORM history specification' in Figure 3, are explained in the next section.

The Code Generator is used to create database procedures in a way to fully utilize provided mechanisms of a selected DBMS. Stored procedures are generated for inserting new element into a concrete table. Firstly, data are inserted into parent tables (keys only), and after that into a corresponding concrete table. Through update procedures, data are updated in concrete tables only. Finally, by deletion procedures, rows are firstly deleted from concrete tables, and then from all its parent tables. By using database stored procedures, we gain on speed, and simplify the way we communicate with the selected DBMS.

By the design of the active database, we provide the possibility to write queries with no JOIN clauses for reads and updates, which is important in accessing

Figure 3

The development process



the latest states in power calculations like load flow. A similar approach is applied in [10], where time performances are significantly improved.

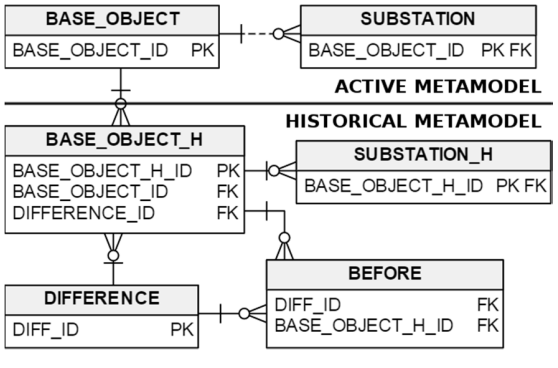
To provide the track of changes on a model instance, restoring the model instances, usage of history data for analysis, statistics or some other calculations, the active model has to be extended. Such extensions are discussed in the next section, in the scope of the second phase of our approach.

5. Historical model

In this section, the second phase of our methodological approach is presented. The historical model (Figure 4) is presented, through which the states of CIM instances are recorded. The historical model is an extension of the active model, where we track changes for concrete tables of the active model. We do not track changes on abstract tables, since all the needed data are in concrete tables. By means of the historical model, we implement a *historical* database.

Figure 4

A database schema of the Historical model



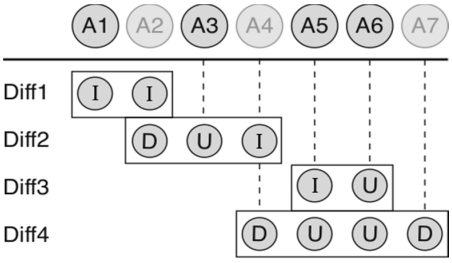
In this phase of our methodological approach, the primary goal is to provide tracking changes of CIM instances, as well as a way to restore model instances to a selected state before some change has been performed. To achieve the goal, this phase of our approach is divided into four steps.

The first step is to create the structure for tracking changes made on a single element from active model. In the second step, we create the structure for tracking changes made simultaneously on a group of elements. After that, we discuss prerequisites to restore a model instance to some previous state in more detail. In the third step, we provide a structure that is used to restore model instances. The final, fourth step is to implement a historical database. In the following text, we elaborate each of the steps in more details.

We describe the historical model, and how it is used for tracking changes. In Figure 4, an excerpt of the active and historical model is shown. For tracking changes on each concrete table from active model, we create a new history table, where we store previous states for each concrete class. For the table that represents the Substation class, a Substation History Table is created (SUBSTATION_H). All history tables have a surrogate primary key from BASE_OBJECT_H table that represents Base Object History class, which has a foreign key from table representing Base Object class. The active and historical models are connected via BASE_OBJECT and BASE_OBJECT_H tables. Base Object History stores information of what action was made, Insert, Update or Delete. Before some action is performed to an element of active database, its state is recoded by creating a new record in corresponding history table. We call those records history elements. Having only history tables, we provide the possibility to track changes efficiently at the level of a sole element only. However, restoring a whole model instance to a particular point in time could be very demanding, as all history tables are to be searched by comparing dates and times.

In the second step, we create a structure that groups changes made simultaneously on a group of elements. We call such a structure difference group, and it represents the difference files, introduced in Section 3. As presented in Figure 4, history tables are connected to DIFFERENCE table via table BASE_OBJECT_H. Each row in DIFFERENCE table represents one difference group. By this, history elements are grouped into difference groups. Figure 5 illustrates the relationship between history elements and difference groups. The elements from active database are shown in circles (A1 to A7), above the horizontal line. Their previous states, i.e. history elements are also shown in circles, placed vertically below the horizontal line. Letters I, U and D, represent actions made on them: Insert, Update and Delete, respectively. Difference

Figure 5
Difference groups



groups (Diff1 to Diff4) are presented with rectangles that group history elements. By grouping history elements into difference groups, it is easier to find changes made simultaneously, in one transaction. With each difference group, we also create a *checkpoint* to which a model instance can be restored. When restoring the model instance by application of a group of changes, we ensure that the model instance will remain in a valid state. For example, for a model instance to remain in a valid state, removing one element from the model instance requires removing all other dependent elements. By this, in order to revert one group of changes, there is no need to search the complete historical database and compare all elements by date and time.

However, to restore the model instance relatively easy and fast it is not enough to have history elements and difference groups only. Following Figure 5, if there is only one group (Diff1), it is easy to restore a model instance to its previous state, as inserted elements need to be removed, the edited items need to be restored, while the ones that are deleted need to be added back. The restoration activity is more complex as we have to follow a longer chain of differences. For example, let us have two difference groups, Diff1 and Diff2. For restoring to a state before Diff1, the changes from Diff1 must be reverted first, and then changes from Diff2 must be reverted, but without overlapping history elements, like in the case of A2. Therefore, we potentially have to compare a large amount of data again to restore the model instance. The more difference groups we have, the longer it takes to restore the model instance.

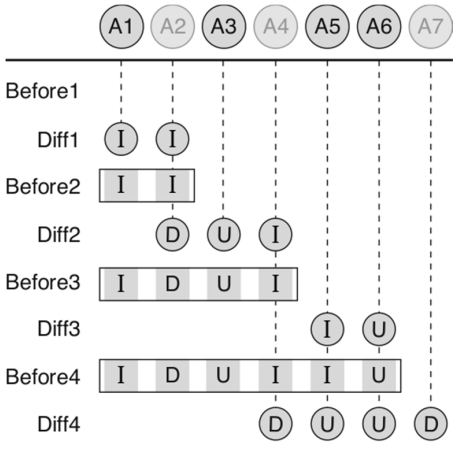
In the third step, we create a structure for reverting model instances. For this, we add *is_first* and *is_last* flags as columns to the table representing Base Object History. With this, for one element from active database, we track which of its history elements is the first and which is the last, respectively. Flag *is_last* has to be updated when new history element is added.

Then, we create a structure that groups all previous last changes from the historical database. We call such a structure before group of changes. In Figure 4, BEFORE table represents before group of changes. Before applying new changes, all history elements with *is_last* flag are recorded into before group, and that way connected to a difference group whose changes are then applied. Flags *is_last* are updated upon applying changes. Each difference group has its before group.

In Figure 6, before and difference groups of changes are shown. Before groups (Before1 to Before4) are presented with rectangles that group key values as references to history elements shown as squares. Here, for difference group Diff3, its before group Before3 consists of a reference to the history element of A1, changed in Diff1, and references to the history elements of A2, A3 and A4, changed in Diff2.

Since there are history elements A5, A6, A7 from Figure 6 that were changed later in Diff3 and Diff4, but those changes were not covered by Before3, we also need to include those elements, in order to restore the model instance to the state before Diff3. Therefore, we introduce after group of changes. It compris-

Figure 6
Before groups in squares



es history elements, which were created after before group of changes was formed. For one before group, its corresponding after group includes all history elements that a) have *is_first* flag, and b) none of their instances are included in the before group. By using those two conditions, we are able to retrieve history elements without the need to search for them by comparing date and time. In Figure 6, for difference group Diff3, after group consists of history elements A5 and A6 changed in Diff3 itself, and A7, deleted in Diff4. The element A4, changed in Diff4, was also changed in Diff2, but contained in Before3, and therefore is not part of after group of changes. The after group is not represented as a separate table structure, as it is defined with the *is_fist* flag and a before group.

At the end of this step, we join corresponding before and after groups of changes into a slice. One slice consists of before changes that are not the last, because their states are current in active database, and after changes that are not contained in before changes. Those are all the needed changes we have to revert in order to restore the model instance before any checkpoint. Each difference group has its slice to undo in order to restore a valid model instance.

In Figure 8, we can see slices presented with rectangles. It is easy to notice that for restoring the model instance to its first (initial) state, it requires the most data to be reverted. However, if we apply slices, less data are needed to revert latest changes.

Once the slice and the model instance are retrieved from relational DBMS, reverting individual changes is the next step. The complete algorithm for reverting a single element of an active model instance is shown in Figure 7. Following the algorithm, in Table 1 we present the possible states of a history element and operations that are to be performed over it. In Table 1,

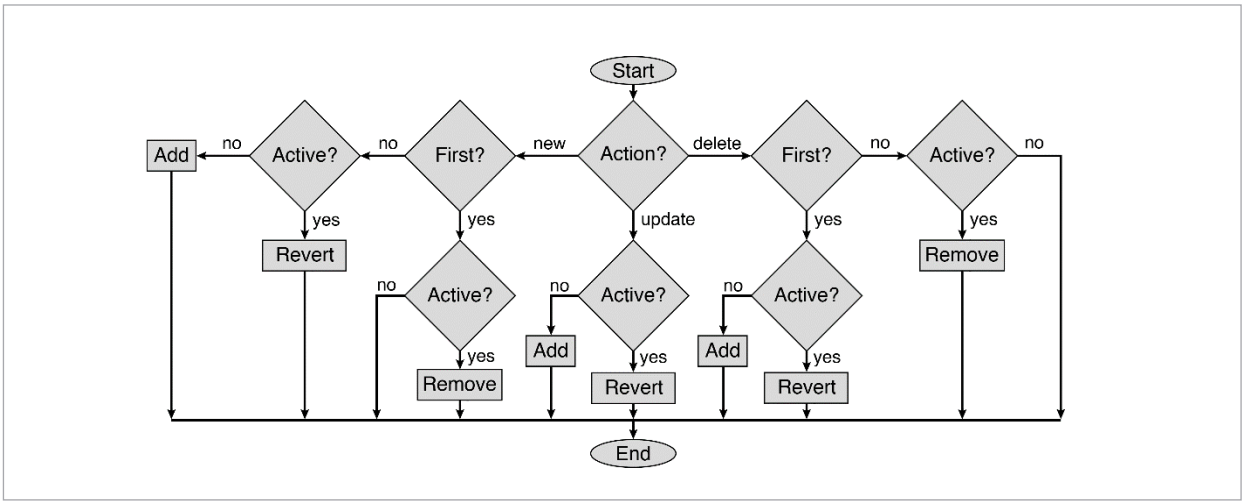
Table 1

Revert algorithm for a single history element

State of a history element			Is first?	Is in active inst.?	Operation to revert			
Insert	Update	Delete			Add	Revert	Remove	Nothing
●			⊤	⊤			√	
●			⊤	⊥				√
●			⊥	⊤		√		
●			⊥	⊥	√			
	●		⊤ or ⊥	⊤		√		
	●		⊤ or ⊥	⊥	√			
		●	⊤	⊤		√		
		●	⊤	⊥	√			
		●	⊥	⊤			√	
		●	⊥	⊥				√

Figure 7

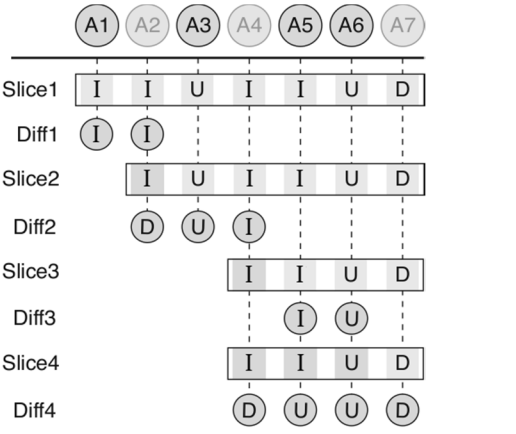
Revert algorithm for a single history element



by dots '●' we mark states of a history element, while '⊤' and '⊥' denote true and false values of decision conditions. The symbol '✓' denotes a selected operation to perform on a history element in order to get restored model instance.

Figure 8

Slices in squares



If a history element is in the Insert action state, its *is_first* flag is set to true, and it exists in the active model instance, the action 'remove' is performed, since the element has been added after a difference checkpoint. If it does not exist in the active model instance, the action 'do nothing' is performed, since the element has been added and deleted after a checkpoint. If the element's *is_first* flag is set to false and the element exists in the active model instance, the action 'revert' is performed, since the element has been added before and changed after a checkpoint. If it does not exist in the active model instance, the action 'add' is performed, as the element has been added before and deleted after a checkpoint.

If a history element is in the Update action state, its *is_first* flag is not of the importance. If it exists in the active model instance, the action 'revert' is performed, since the element has been edited either before or after checkpoint. If it does not exist in the active model instance, the action 'add' is performed, since the element has been edited either before or after, and deleted after checkpoint.

If a history element is in the Delete action state, its *is_first* flag is set to true, and it exists in active model instance, the action 'revert' is performed, since the element has been firstly deleted after slice and then element with the same ID was added. If it does not exist in active model instance, the action 'add' is performed, since the element has been deleted after checkpoint. If its *is_first* flag is set to true, and the element exists in active model instance, the action 'remove' is performed, since the element has been deleted before and added after checkpoint. If it does not exist in the active model instance, the action 'do nothing' is performed, since the element has been deleted before, added back and deleted again.

In the final, fourth step, we present the implementation of the second phase of our approach. Here for all 200 concrete classes of the CIM model we need to create their history classes, map them to the tables, and create stored procedures and the appropriate API communication layer to approach them. By our practical experience, it requires high implementation effort. Therefore, we have applied our Code Generator, developed to support the implementation. As the historical model is an extension of the active model, we also extend the Code Generator to create a historical database.

A development process covered by the second phase of our approach is also presented in Figure 3. The Code Generator reads the CIM UML specification and generates the object model comprising C# CIM history classes. An XML ORM history specification is created in accordance to the approach already presented in the previous step. Finally, a historical database is created by using the NHibernate ORM framework.

The Code Generator is used to produce database procedures that create history elements in a historical database, before any action is applied on the active database. Those procedures contain the calls to the active database procedures, explained in the previous section. In this way, we ensure that all changes on a CIM instance are tracked and can be reverted upon a request.

In this section, we have presented a new approach for storing changes made on CIM instances. It allows us to restore the model instance before any wanted state. To the best of our knowledge, this is a novel approach, which introduces improvements in a design of a database to store model instances, track changes on a model instance and restoring it.

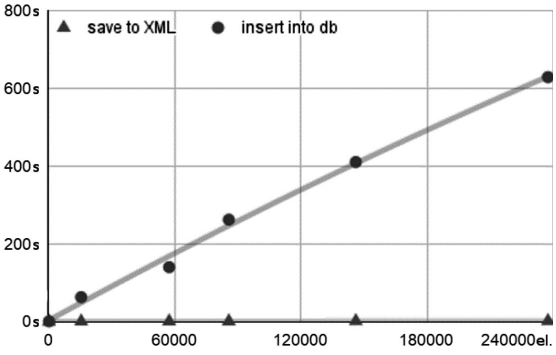
6. Evaluation

To support the active and historical CIM models, we have developed the active and historical relational databases. Tracking changes in the historical database increases complexity. Therefore, time performances may drop. Here we analyze how complexity affects our relational database. Then, we discuss the usage of our Code Generator. We present diagrams, where horizontal axis shows a number of model elements, while vertical axis shows time of performed operations over elements, in seconds. The tests were done on a PC, CPU Intel Core 2 Duo E7500 2.93GHz, with 8GB RAM. Oracle Database 11g Express Edition was used as a relational DBMS.

The operations over the elements in the active model are the most frequent. By testing the system and calculating average time for reading and inserting active model instances, we created diagrams shown in Figure 9 and Figure 10, respectively. In Figure 9, we have two functions displayed, for average time needed for reading an active model instance from XML file (a line with triangles) and from the active database (a line with circles). The time needed to complete the task linearly depends on the number of elements. Reading active model instances is not affected by amount of historical data. Inserting CIM instance into a database is also linearly dependent, as it is presented in Figure 10. For model instances, with number of elements ranging from 500 up to 240.000, required time is from 1.7 up to 620 seconds. However,

Figure 10

Inserting a model into database and XML



in comparison to saving it as XML files, it is noticeably slower, where required time is from 1.2 up to 2.5 seconds, for the same instances. This is due to the fact that saving to XML files is done by serialization, while storing to database follows a complex table structure, as explained in Section 4.

As every change made on a model instance is tracked, for every operation we have one more action for recording a before state. Thus we create a before group of changes. On average, changing a model instance, with tracking history, requires only 65% more time. Figure 11 shows the performance ratio when modifying a model instance with and without tracking changes, which is presented with circles and trian-

Figure 9

Reading a model from XML and from database

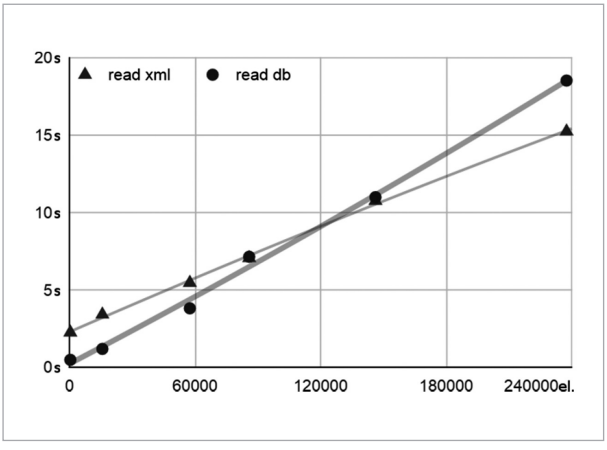
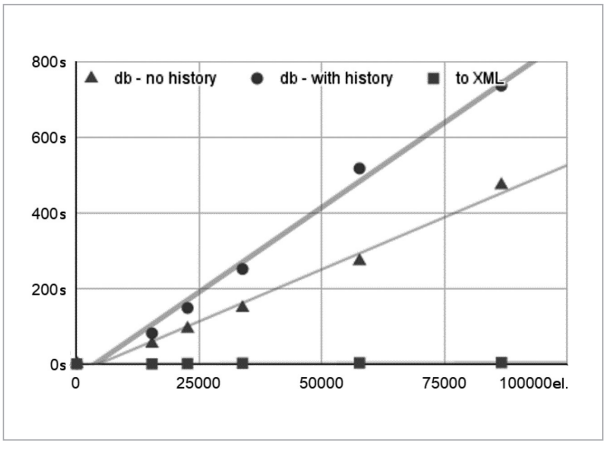


Figure 11

Making changes with and without history



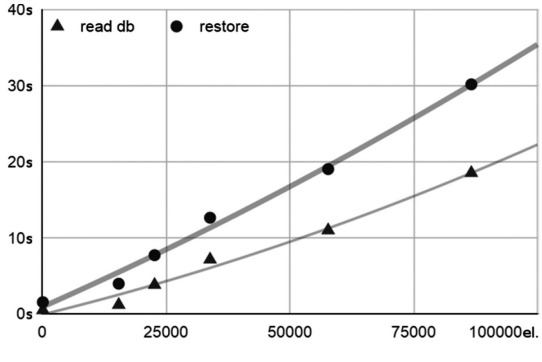
gles, respectively. For saving changes to database, with tracking history, ranging from 170 up to 86.000 elements, required time is from 2.1 up to 740 seconds. Without tracking history, required time is from 1 up to 475 seconds, for the same instances. When saving to XML, required time is from 1.2 up to 4.2 seconds.

We have obtained such result by the application of our generated stored procedures for maintaining historical model instance while working with active model instance, as presented in Section 5. In this way, we are using full capabilities of the relational DBMS. In our experiments, on each model instance, various changes have been made, and around 45% of all elements in a model instance have been affected. When saving CIM instance to the XML files, after applying changes, performance is similar to that from Figure 10, because there is no tracking of changes.

To restore a model instance, as we explained in Section 5, we must have a slice of changes. In Figure 12, we present the results of revert operations applied on a model instance, by means of the selected slice. Time needed to read the current model instance is shown with triangles, while time for its restoring is shown with circles. Firstly, the current model instance is read from active database, secondly a needed slice is read, and finally the slice is applied on the model instance – model is restored to a checkpoint of that slice. On average, after reading the model instance, 75% more time is needed to complete the restoration. This is possible because the time is not lost on searching for changes to revert, but simply reading the changes from the slice representing a corresponding checkpoint.

Figure 12

Reading from database and restoring the model



Now, we observe a number of lines of code being generated. In many examples, as it is in [18] and [2], it is provided a generation of 48% and 75% of total lines of code for their general-purpose solutions. We provide only a generation of API communication layer and database functions, and around 95% of total lines of code is generated. If we would consider other application components, this number would be smaller. Nonetheless, we consider this as a good result, since there is a need to provide an efficient code generator in support of our methodological approach, which is in many aspects relatively complex.

7. Conclusion and Future work

In this paper, we present an approach to implementation of CIM instance storage in a relational database system, in order to provide efficient executions of operations over model instances stored in the database. In our approach, we support both active and historical models of power grids. Thus, track change and restore operations, as complex and demanding in practice, are fully supported. In addition, we have developed a code generator to support an easy and efficient adaptation of the implemented database system to constantly emerging CIM changes. Finally, we have evaluated our system by measuring times needed to perform operations of saving a CIM instance, its reading, changing (with and without tracking changes), and restoring. By this, we believe that we may contribute to a wider acceptance of CIM in power grid networks.

Our future research will include: (i) development of a data warehouse system for reporting and data analysis; (ii) improving the code generator to include a component for graphical representation of CIM instances and all states of CIM instances stored in the database system; and (iii) improving a code generator to provide a wider selection of implementation platforms, including various DBMSs.

Acknowledgments

The research presented in this paper was supported by the Ministry of Education, Science, and Technological Development of the Republic of Serbia under Grant III-44010.

References

1. Ablonskis, L., Nemuraitė, L. Discovery of Model Implementation Patterns in Source Code. Information Technology and Control, 2010, 39(1), 68-76.
2. Antović, I., Vlajić, S., Milić, M., Savić, D., Stanojević, V. Model and Software Tool for Automatic Generation of User Interface Based on Use Case and Data Model. IET Software, 2012, 6(6), 1-15.
3. Armonas, A., Nemuraitė, L. Pattern Based Generation of Full-Fledged Relational Schemas from UML/OCL Models. Information Technology and Control, 2006, 35(1), 27-33.
4. Atkinson, C., Kuhne, T. Model-Driven Development: A Metamodeling Foundation. IEEE Software, 2003, 20(5), 36-41. https://doi.org/10.1109/MS.2003.1231149
5. Bezivin, J., Gerbe, O. Towards a Precise Definition of the OMG/MDA Framework. Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE), 2001, 273-280. https://doi.org/10.1109/ASE.2001.989813
6. Britton, J. P., Brown, P., Moseley, J., Bunda, M. Optimizing Operations with CIM: Today's Grid Relies on Network Analysis (and a Lot of Data). IEEE Power and Energy Magazine, 2016, 14(1), 48-57. https://doi.org/10.1109/MPE.2015.2481783
7. Britton, J. P., deVos, A. N. CIM-Based Standards and CIM Evolution. IEEE Transactions on Power Systems, 2005, 20(2), 758-764. https://doi.org/10.1109/TPWRS.2005.846202
8. Capko, D., Erdeljan, A., Vukmirovic, S., Lendak, I. A Hybrid Genetic Algorithm for Partitioning of Data Model in Distribution Management Systems. Information Technology and Control, 2011, 40(4), 316-322. https://doi.org/10.5755/j01.itc.40.4.981
9. Darwen, H. The Relational Model: Beginning of an Era. IEEE Annals of the History of Computing, 2012, 34(4), 30-37. https://doi.org/10.1109/MAHC.2012.50
10. Dević, S., Atlagić, B., Gorečan, Z. Database Modelling and Development of Code Generator for Handling Power Grid CIM Models. ICEST Conference, 2011.
11. ENTSO-E. Common Grid Model Exchange Standard (CGMES) – Based on IEC Common Information Model, version 2.4, August 2014.
12. Fertalj, K., Kalpic, D., Mornar, V. Source Code Generator Based on a Proprietary Specification Language. Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS), 2002. https://doi.org/10.1109/HICSS.2002.994498
13. Hazzard, K., Bock, J. Metaprogramming in. NET. Manning Publications, 2013.
14. Henthorne, C., Tilevich, E. Code Generation on Steroids: Enhancing COTS Code Generators Via Generative Aspects. Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS '07), 2007.https://doi.org/10.1109/IWICSS.2007.4
15. IEC 61970 Energy Management System Application Program Interface (EMS-API) – Part 301: Common Information Model (CIM) Base. IEC, Edition 2.0, 2007.
16. Ivanov, C., Chury, D. European Electric Power System on the Way Towards Implementation of CIM Based Data Exchange Format. IEEE Power & Energy Society General Meeting (PES '09), 2009.
17. Koss, A. M. Programming on the Univac 1: A Woman's Account. IEEE Annals of the History of Computing, 2003, 25(1), 48-59. https://doi.org/10.1109/MAHC.2003.1179879
18. Kundu, D., Samanta, D. Mall R. Automatic Code Generation from Unified Modelling Language Sequence Diagrams. IET Software, 2013, 7(1), 12-28. https://doi.org/10.1049/iet-sen.2011.0080
19. Leavitt, N. Will NoSQL Databases Live Up to Their Promise? Computer, 2010, 43(2), 12-14. https://doi.org/10.1109/MC.2010.58
20. Lendak, I. I., Erdeljan, A. M., Popović, D. S. Algorithm for Cataloging Topologies in the Common Information Model (CIM). Computers and Mathematics with Applications, 2011, 61(3), 715-721. https://doi.org/10.1016/j.camwa.2010.12.021
21. Levy, L. S. A Metaprogramming Method and Its Economic Justification. IEEE Transactions on Software Engineering, 1986, SE-12(2), 272-277. https://doi.org/10.1109/TSE.1986.6312943
22. Ma, K., Yang, B., Abraham, A. A Template-Based Model Transformation Approach for Deriving Multi-Tenant SaaS Applications. Acta Polytechnica Hungarica, 2012, 9(2), 25-41.
23. Nayak, A., Poriya, A., Poojary, D. Type of NOSQL Databases and Its Comparison with Relational Databases. International Journal of Applied Information Systems, 2013, 5(4), 16-19.

24. Nedić, N., Švenda, G. Workflow Management System for DMS. Information Technology and Control, 2013, 42(4), 373-385. https://doi.org/10.5755/j01.itc.42.4.4546

25. Nguyen, V. C., Qafmolla, X., Richta, K. Domain Specific Language Approach on Model-Driven Development of Web Services. Acta Polytechnica Hungarica, 2014, 11(8), 121-138.

26. Pradeep, Y., Seshuraju, P., Khaparde, S. A., Joshi, R. K. CIM-Based Connectivity Model for Bus-Branch Topology Extraction and Exchange. IEEE Transactions on Smart Grid, 2011, 2(2), 244-253. https://doi.org/10.1109/TSG.2011.2109016

27. Ravikumar, G., Khaparde, S. A., Pradeep, Y. CIM Oriented Database for Topology Processing and Integration of Power System Applications. Power and Energy Society General Meeting (PES), 2013. https://doi.org/10.1109/PESMG.2013.6672164

28. Schulte, S., Berbner, R., Steinmetz, R., Uslar, M. Implementing and Evaluating the Common Information Model in a Relational and RDF-Based Database. Information Technologies in Environmental Engineering, Environmental Science and Engineering, 2007, 109-118. https://doi.org/10.1007/978-3-540-71335-7_13

Summary / Santrauka

The ongoing development of a complex model for power grid networks, based on the Common Information Model (CIM), is dealing with design, operability and exchange of data among various power grid operators. This paper presents a methodological approach to development of a database that supports an easy storing and managing of active CIM instances, as well as their historical versions. To facilitate the implementation of the proposed approach, we apply a paradigm of automatic programming. Our code generator eases the work on developing an API communication layer over the database and allows faster response on CIM changes. Finally, we present a performance cost analysis on test models. By this, our intention is to contribute to a wider acceptance of CIM in power grid networks.

Nenutrūkstamas kompleksinio elektros energijos tinklų modelio vystymas, pagrįstas Bendruoju informacijos modeliu (CIM), siekia patobulinti modelio dizainą ir veiksmingumą bei duomenų keitimąsi tarp energijos tinklų operatorių. Straipsnyje pristatomas metodologinis duomenų bazės, palaikančios nesudėtingus aktyvių CIM kaupimo ir valdymo atvejus ir jų istorines versijas, sukūrimo būdas. Siekdami palengvinti siūlomo metodo įgyvendinimą, autoriai taiko automatinio programavimo paradigmą. Jų kodų generatorius palengvina darbą kuriant API ryšių sluoksnį per duomenų bazę ir leidžia greičiau reaguoti į CIM pokyčius. Galiausiai pristatoma ir bandomųjų modelių našumo sąnaudų analizė. Straipsnio autoriai siekia prisidėti prie platesnio CIM taikymo elektros energijos tinkluose.