Received: 26 August 2021 | Revised: 23 September 2022 | Accepted: 9 October 2022

DOI: 10.1049/tje2.12208

The Journal of Engineering

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH

Template-based generation of programming language specific code for smart grid modelling compliant with CIM and CGMES

Jan Dinkelbach[1] | Lukas Razik[2] | Markus Mirz[1,3] | Andrea Benigni[2,4] | Antonello Monti[1,3]

[1]Institute for Automation of Complex Power Systems, RWTH Aachen University, Aachen, Germany

[2]IEK-10: Energy Systems Engineering, Jülich Research Centre, Wilhelm-Johnen-Straße, Jülich, Germany

[3]Center for Digital Energy Aachen, Fraunhofer FIT, Aachen, Germany

[4]Chair of Methods for Simulating Energy Systems, RWTH Aachen University, Aachen, Germany

Correspondence
Jan Dinkelbach, Institute for Automation of Complex Power Systems, RWTH Aachen University, Mathieustr. 10, Aachen, Germany.
Email: jdinkelbach@eonerc.rwth-aachen.de

Funding information
H2020 Leadership in Enabling and Industrial Technologies, Grant/Award Number: I-NERGY - Grant agreement ID: 101016508; Helmholtz Association, Grant/Award Number: Joint Initiative Energy Systems Integration; H2020 Energy, Grant/Award Number: SOGNO - Grant agreement ID: 774613

Abstract

The transition to Smart Grids increases the complexity of power grids by involving many more interdependent actors and integrating additional information and communications technology. To provide a common basis for Smart Grid data representation and exchange, the standardized Common Information Model (CIM) has been introduced and extended, i. a., by the Common Grid Model Exchange Specification (CGMES). An increasing acceptance by power grid operators and other actors has made CIM and CGMES more and more relevant. However, the implementation of CIM / CGMES support in software projects appears to be challenging due to the complexity of CIM / CGMES and the ongoing standardisation process with iterative adaptations. Thus, the main contribution of this paper is the presentation of a methodology for an automated generation of programming language specific code from CIM / CGMES specifications. The approach is based on the use of a template language and enables to keep software projects fully compliant with CIM / CGMES specifications. The paper outlines the process of code generation and the consecutive codebase integration for a JavaScript based CIM / CGMES web editor and for two CIM / CGMES de-/serialiser libraries in C++ and Python. The approach is evaluated in use cases involving the visualisation and simulation of a benchmark grid.

1 | INTRODUCTION

1.1 | Motivation

The transition from conventional power grids to Smart Grids requires advanced grid monitoring and control. This is accompanied by the incorporation of many more interdependent actors and the integration of additional information and communications technology (ICT) systems. The corresponding increase in information exchange demands efficient communication technologies as well as ensuring the security and consistency of the data transfer [1]. Recent technological trends like 5G communication and machine learning may enable an efficient communication and secure data transfer [2–4]. Still, ensuring the consistency of the information exchange between a variety of actors and heterogeneous ICT systems remains a key challenge in Smart Grids.

As an example, the installation of renewable energy sources (RES) needs a coordinated operation of transmission and distribution systems and, correspondingly, an increasing data exchange between transmission system operators (TSOs) and distribution system operators (DSOs). This should be based on a standardized data representation and exchange, for example, implemented by energy management systems (EMSs) of TSOs [5] and distribution management systems (DMSs) of DSOs [6]. Here, not only the syntax but also the semantics of data representation and exchange must be explicitly defined to avoid ambiguities and misunderstandings,

achieving compatible implementations coming from different entities.

In this context, the Common Information Model (CIM) was developed to meet the requirements of a well-defined and extensible data model, which is nowadays used by many entities in the energy sector [7]. The base model of CIM has been defined by the International Electrotechnical Commission (IEC) as part of the standard IEC 61970 [8]. Targeting the energy transmission management, the base model enables in particular the representation of power grid topologies and their assets. Besides, CIM was extended within the standard IEC 61968 for distribution management [9] and within IEC 62325 for energy market communications [10]. CIM belongs to IEC's core semantic standards for Smart Grids, as well as IEC 61850 [11]. Further CIM use cases are the system integration using predefined interfaces between the CIM of DMSs and automation parts as well as custom system integration using XML-based payloads for semantically sound coupling of systems [12]. Besides, CIM has been employed as a base data model in multi-domain co-simulations for Smart Grid planning [13].

The Common Grid Model Exchange Specification (CGMES) is a superset of the CIM based data exchange standard (Profile 1 based on CIM UML `14v02`) of the European Network of Transmission System Operators for Electricity (ENTSO-E) and is specified by IEC TS 61970-600 [14]. For instance, additionally to CIM, CGMES defines the file exchange and extensions to CIM classes. Its purpose is the interface definition for TSO software in order to exchange power grid modelling information as required by the ENTSO-E and TSO business processes. The TSOs use it, for example, for modelling in power flow and contingency analysis, short circuit calculations, and dynamic security assessment.

A substantial challenge is that CIM and CGMES are undergoing frequent changes during the transition to Smart Grids, for example, with CGMES versions v2.4.13, v2.4.14, and v2.4.15 released in 2013, 2014, and 2015 respectively [15]. Besides, the specified data models contain hundreds of classes and attributes. This makes it difficult to keep the source code of CIM / CGMES based software projects constantly up-to-date and fully compliant with the CIM / CGMES specifications. In fact, manual software updates would be very cumbersome and error prone. Hence, the frequent changes and the complexity of CIM / CGMES motivate the application of an automated source code generation approach for CIM / CGMES based software projects.

1.2 | Contribution and outline

The main contribution of this paper is the presentation of an automated generation of programming language specific code from CIM / CGMES specifications. The concept presented in this work is based on the Mustache template language [16] and has been implemented in the software project *CIMgen*. CIMgen incorporates our crucial methodology to perform the automated code generation and ensure that the processing of CIM / CGMES based data can be kept compliant with the frequently changing and complex CIM / CGMES specifications. The generated code, obtained by using CIMgen, has been integrated into the recently released software projects *CIMpy* and *Pintura*, both presented in this paper for the first time, as well as into the existent software project *CIM++* [17] and evaluated in use cases covering network visualisation and simulation.

Within the scope of this paper, we begin with the presentation of related work carried out regarding the automated code generation for CIM / CGMES in Section 2. Then, Section 3 explains relevant fundamentals and, based on them, Section 4 introduces the proposed concept of language-specific code generation from CIM / CGMES specifications. In Section 5, we outline how we address the different programming language characteristics and explain the code generation process for a JavaScript codebase required in a graphical power grid topologies editor as well as for two codebases of de-/serialiser libraries written in C++ and Python. Finally, Section 6 evaluates the generated codebases applying them in use cases which consider the visualisation and simulation of a benchmark grid.

2 | RELATED WORK

The here presented code generation belongs to the forward engineering approaches of the traditional software engineering disciplines [18]. Besides the methodology proposed in this paper, the only such approaches for CIM known to the authors have been employed for the software projects CIM++ and PyCIM (the latter is not to be confused with the here presented CIMpy).

In case of CIM++, an automated deserialiser generation was presented in [17], which is based on a C++ codebase generation from CIM UML with the aid of a UML editor. A succeeding correction, extension, and adaptation of the CIM C++ codebase is needed as, for instance, the CIM UML defines an `Integer` class but not how this shall be implemented in a particular programming language. The general usability of the generated codes out of CIM / CGMES specifications has been presented in case of CIM++, for example, in [19], for a converter from CIM based topologies to power grid simulator specific system models.

In contrast to the proceeding in [17], the methodology proposed in this paper does not require any adaptation steps after the generation of the CIM C++ codebase, as all the needed information is used during the code generation by the presented source code generator CIMgen. Beyond that, depending on the templates, CIMgen can output program code for different languages and does not rely on suitable compiler frontends for further adaptation of the CIM codebase, as, for example, it was the case for the CIM C++ codebase of CIM++. More than that, CIMgen is completely freely available under an open source license [20] and independent from further software tools such as commercial UML editors like, for example, Enterprise Architect that was needed by CIM++.

With *PyCIM* there is a CIM implementation in Python [21]. Its code generation from CIM XMI is sketched in [22] which makes use of a toolchain around the meta model *Ecore* as intermediate representation. The needed tools are not maintained

anymore and the original developer considered in August 2017 to mark PyCIM as deprecated.

With CIMgen we avoid the usage of intermediate representations and several tools. Instead, we generate the codes directly from the provided documents specifying CIM and CGMES. Consequently, there is no need for maintenance of several tools. CIMgen supports the generation from the freely available documents provided by ENTSO-E which makes CIMpy and CIM++ compatible with CGMES, whereas in PyCIM the support for CIM16 and higher versions as well as CGMES may be available under a contract only [21]. With respect to PyCIM, the here presented CIMpy introduces fundamental changes with the incorporation of CIMgen's codebase, due to which we refrained from substantially modifying the existent PyCIM project.

3 | FUNDAMENTALS

For the understanding of the overall concept, several fundamentals related to CIM / CGMES and the proposed concept of this paper are explained in the following. Our concept is based on the processing of machine-readable documents that specify CIM / CGMES. The related specification languages and text-based formats that are mainly employed by the institutions specifying CIM / CGMES are briefly explained. Thereafter, we introduce the concept of template processing by means of a template engine, as our automated source code generation makes use of it.

3.1 | CIM and CGMES

The CIM is maintained by the CIM User Group (CIMug) based on the Unified Modeling Language (UML), a formalism for graphical object-oriented modelling [23]. With UML class diagrams, object classes and relations between them (e.g. *inheritance*, *associations*, *aggregations* etc.) can be defined. CIM UML specifies which kind of objects a CIM document can contain and how these objects are interlinked, which more generally spoken is called an *ontology* [24]. The UML class diagrams contain attributes only, as there is no need for function definitions in case of CIM as information model specification. The different CIM UML versions are distributed in machine-readable form, for example, as XMI documents. A more comprehensive introduction to CIM is provided by [25].

Instead, CGMES is maintained by the ENTSO-E in the CGMES Library which contains all documents approved by ENTSO-E respectively IEC for a harmonisation and implementation of standards related to CGMES [15]. For the structure description of CGMES documents, there are so-called RDFS documents, provided by ENTSO-E.

Both RDFS documents (for CGMES) and XMI documents (for CIM) form the basis of our automated code generation approach. Therefore, we introduce their fundamentals in the following.

3.2 | XML and XSD

The Extensible Markup Language (XML) is a well established markup language, standardised by the World Wide Web Consortium (W3C) for human- and machine-readable documents [26]. XML works with *tags* that are not predefined and therefore can be invented by the editor of the XML file. For instance, the attributes of a book could be stored as follows:

```
<book>
  <title>Stories</title>
  <author>John Doe</author>
</book>
```

The problem of a sole XML document is that syntax and semantics of the tags (e.g. `<book>`) are not formally defined and that only hierarchical (i.e. tree-like) structures can be represented.

With the aid of the XML Schema Definition (XSD) language, the structure and content of XML documents can be described by a determined syntax which defines the elements and attributes [27]. For the upper example this XSD document could look as follows:

```
<xs:element name="book">
  <xs:complexType>
      <xs:sequence>
        <xs:element name="title"
            type="xs:string"/>
        <xs:element name="author"
            type="xs:string"/>
      </xs:sequence>
  </xs:complexType>
</xs:element>
```

However, XSDs still do not define the semantics. This issue is addressed by the Resource Description Framework (RDF).

3.3 | RDF and RDFS

The RDF describes *resources* (i.e. arbitrary entities) and their relations [28, 29]. RDF documents consist of statements (also called *triples*) with a

- *subject*, the resource to be described;
- *predicate*, the *property* of the resource;
- *object*, the value of the property.

The RDF is an abstract model whose syntax respectively document's format is not determined. Possible formats are N3, Turtle, RDF/XML, graphs, and so forth [29].

The *identifiers* of resources should be unique. Therefore, Uniform Resource Identifiers (URIs) are often used as they can point to the worldwide unique location of a resource.

Similarly to XSD, in case of XML documents, RDF Schema (RDFS) provide a data modelling *vocabulary* for RDF documents [30]. With the vocabulary a particular application domain can be modelled. RDFS therefore allows to formalise simple

ontologies for complexity limitation and organisation of data into information and knowledge. Important RDFS constructs are: `rdfs:class` for class definitions, `rdfs:subClassOf` for inheritance representation, `rdfs:domain` for a class declaration of the subject in a triple, and `rdfs:range` for a class or type declaration of the object. An example ontology could be expressed with the aid of RDFS as follows:

```
<rdf:RDF ...>
  <rdfs:Class rdf:ID="Person">
    <rdfs:comment>Class of all human beings
      </rdfs:comment>
  </rdfs:Class>
  <rdfs:Class rdf:ID="Student">
    <rdfs:comment>A person who likes to learn
      </rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Person"/>
  </rdfs:Class>
  <rdfs:Class rdf:ID="Book"/>
  <rdf:Property rdf:ID="reads">
    <rdfs:domain rdf:resource="#Student"/>
    <rdfs:range rdf:resource="#Book"/>
  </rdf:Property>
</rdf:RDF>
```

A corresponding RDF/XML document (i.e. instantiation of the RDFS) could be:

```
<rdf:RDF ... xmlns:school="http://city.tld/
    school.rdf#">
  <school:Student rdf:about="http://city.tld/
      people/1234">
    <school:name>Bart Simpson</school:name>
    <school:reads rdf:resource="http://city.
      tld/library/faust"/>
  </school:Student>
  <school:Book rdf:about="http://city.tld/
      library/faust">
    <school:name>Faust</school:name>
    <school:author>Johann Wolfgang Goethe
      </school:author>
  </school:Book>
</rdf:RDF>
```

The elements `name` and `author` would be defined in the `school` *namespace*. More on RDF/XML can be found in [31]. Such RDF/XML documents are used for both, CIM and CGMES documents. They store the actual CIM / CGMES objects which are the instances of the classes as specified by CIM / CGMES. The process of reading in objects from RDF/XML into the main memory is called *unmarshalling* performed by a *deserialiser*. The other way round is called *marshalling* and performed by a *serialiser*.

RDFS enables the description of metadata with declarative semantics which allows statements on classes and properties. The modelling with the aid of RDFS is possible in many application areas such as in the energy sector. Therefore, RDFS documents for the modelling of CGMES are provided by the ENTSO-E [15]. Additionally, a format used for storing CIM UML specifications is introduced in the following.

3.4 | XMI

The XML Metadata Interchange (XMI) is a standard of the Object Management Group (OMG) which enables an open and vendor-neutral data exchange of objects in terms of XML elements and attributes according to the Meta Object Facility (MOF) [32]. MOF closes the gap between different metamodels by creating a common basis for them. Besides UML models, arbitrary metadata can be exchanged using XMI as long as it can be expressed by the MOF. Among others, the XMI standard defines mechanisms to link objects within the same document and across documents, the validation of XMI documents, and how objects can be referenced (e.g. using Universally Unique Identifier (UUID)).

The CIM UML drafts provided by the CIMug can be stored as XMI files and can be read by standard XML parsers which allows a simple storage, transfer, and processing. A source code generation from UML diagrams is called *forward engineering*. This can be done, for example, by a UML editor with code generation capability or by a separate code generator from an XMI. To summarise the introduced fundamentals, the relationships of the presented documents to each other and to CIM / CGMES UML are shown in Figure 1.

3.5 | Template engine

The automated source code generation presented here utilises a *template engine* (also called *template processor* or *template system*) as it is common, for example, in web site development. Template engines allow the separation of the *model* (i.e. logic and data) and the *view* (i.e. resulting code). For the source code generation it means that the Python source code of CIMgen contains no code that is written by CIMgen into the generated resulting documents. To achieve this, template engines have a

- *data model*: for instance based on a database, a text / binary file, or a container type of the template engine's programming language,
- *template files* (also called *templates*): written in the language of the resulting documents together with special *template language* statements, and
- *result documents*: which are generated after the processing of data and template files, so-called *expanding*.

A simple template file with the C++ statement

```
cout << "Hello {{name}}!" << endl;
```

containing a place holder `{{name}}` can be substituted by the name `World`, for instance from a dictionary from which the template engine gets various names. The resulting C++ would look as follows:

```
cout << "Hello World!" << endl;
```

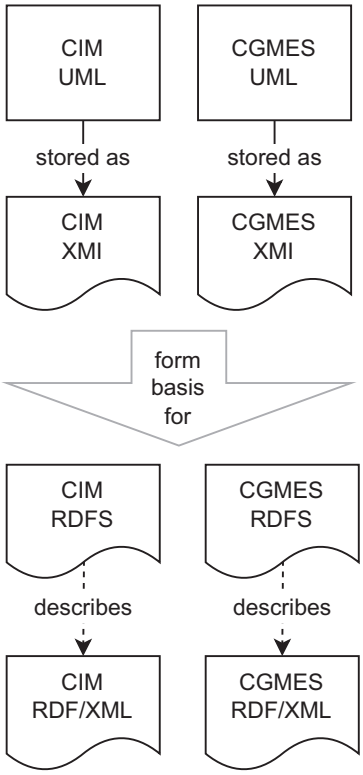

FIGURE 1 The CIM / CGMES UML can be stored in the form of an XMI document. It forms the basis for CIM / CGMES RDFS documents which describe the structure of CIM / CGMES RDF/XML documents. Smart Grid topologies are then stored as CIM / CGMES RDF/XML documents that are structured as required by the associated RDFS documents

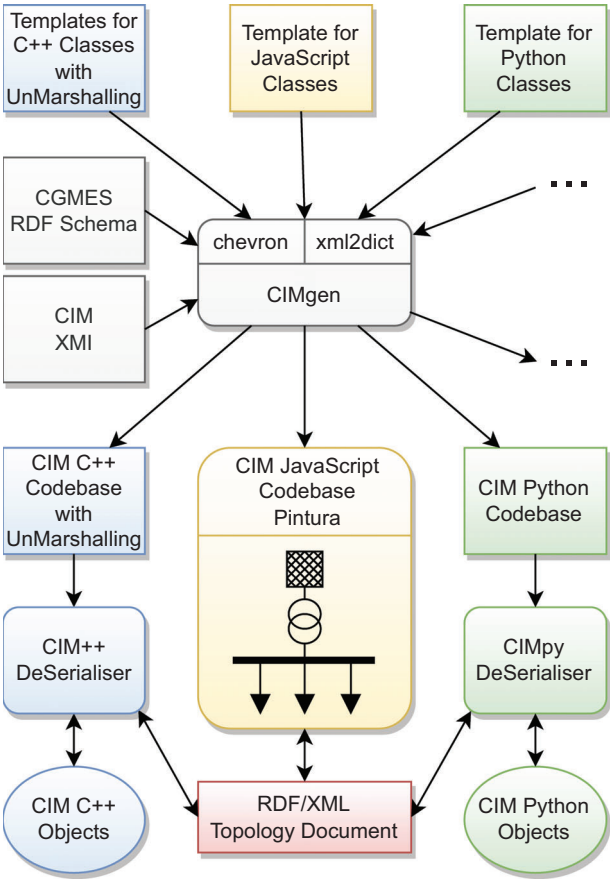

FIGURE 2 CIMgen serves as source code generator for software projects compliant with CIM / CGMES specifications. The source code generated by means of CIMgen forms the basis of software projects like the graphical web editor Pintura and the de-/serialiser libraries CIM++ and CIMpy

For CIMgen the Python project *chevron* was chosen as template engine [33]. Chevron works independently from the language in which the templates are written. For chevron these are just strings containing *tags* of the *Mustache* template language which can be, i. a., of following types:

- *Variable* tags of the form `{{VAR}}` which are replaced by text.
- *Begin* and *End Section* tags of the form `{{#SECTION}}`…`{{SECTION}}`, enclosing sections which may appear zero to N times.
- *Partial* tags of the form `{{>SUBTEMPLATE}}`, advising chevron to insert and expand another template (*subtemplate*) at the tags' location, which also can be repeated zero to N times in the output.

Data are to be provided for the mappings from *keys* (e.g. `name`) to *values* (e.g. `World`). More on Mustache and the substitution process of tags by data, called *rendering*, can be found in [16]. In case of CIMgen the data comes from CIM / CGMES specifications. The concept of this approach is presented in more detail in the following.

4 | CONCEPT

The overall concept of the CIM / CGMES to language-specific code generation is shown in Figure 2. The main two Python libraries used by the source code generator CIMgen are the template engine chevron and *xml2dict* as converter between XML strings and Python dictionaries. Python was chosen as programming language for CIMgen as it is one of the most popular scripting languages due to its simplicity. This makes CIMgen easily maintain- and extendable. Conceptually, CIMgen reads in RDF Schema or XMI files specifying a certain version of CIM / CGMES and renders templates written in arbitrary formats according to the CIM / CGMES specification.

In this work, CIMgen is applied on templates addressing several software projects written in different programming languages: the web based graphical CIM / CGMES editor *Pintura* as well as the two de-/serialiser libraries CIM++ and CIMpy. CIMgen is employed for the generation of the CIM JavaScript, CIM C++ and CIM Python codebases respectively during the software projects' implementation. The hereby implemented forward engineering allows an automated update of these and other software projects whenever CIM / CGMES changes and avoids cumbersome and error prone manual adaptations of the source codes. The code generation process for these projects will be explained in the following.

4.1 | Pintura

The aim of Pintura [34] is to provide a human-machine interface for CIM / CGMES. While RDF/XML documents are easily processed by machines, they are not a good representation for the user to understand and edit grid data. Pintura allows the user to explore and edit grid data through a graphical user interface (GUI). After editing, Pintura stores the grid again in CIM / CGMES so that it can be processed by CIM / CGMES compatible applications, for example, using the de-/serialiser libraries CIM++ and CIMpy. Since web browser based graphical user interfaces support cross-platform compatibility, we decided to build Pintura as a web application relying on HTML, CSS, and JavaScript.

CIMgen employs a template to generate JavaScript classes according to the CIM / CGMES specification. The JavaScript codebase enables to handle the CIM class instances when the grid data is loaded. For the graphical representation, the network diagram is generated as a SVG graphic, which can be easily exported and imported in other applications. This feature has already been used to visualise grid data in the co-simulation web frontend VILLASweb, which is part of the VILLASframework [35].

4.2 | CIM++

The CIM++ [36] de-/serialiser library *libcimpp* is written in C++. The original code generation for libcimpp is presented in [17]. It is based on a first generation stage of class header and implementation files, which form the CIM C++ codebase, as well as on a subsequent generation of the actual unMarshalling code for libcimpp. In case of CIMgen, these two generation stages are combined to one as presented later in this paper. The library is of particular interest for performance critical applications written in C++, for example, for a C++ real-time simulator for power grids.

4.3 | CIMpy

CIMpy [37] is a de-/serialiser library written in the Python language. It additionally features functionalities to modify grid data compliant with CIM / CGMES. The motivation behind the Python library is that engineering solutions, for example, for grid monitoring or control take often grid model information into account while being implemented in Python.

5 | IMPLEMENTATION

In this section, we explain the implementation of the generation of language specific code based on the use of templates. We explain how we address programming language characteristics and how the generated code is integrated into the three different software projects. In the following, CIM always stands for CIM and CGMES.

5.1 | Pintura

For the visualisation of CIM data with Pintura, CIMgen generates a JavaScript class for each CIM class based on the *class file template* depicted in Listing 1. Each JavaScript class has three static functions to render CIM data. Here, only the most important function `attributeHTML` is presented. This function is composed of several sections following the same structure, one section per CIM class attribute. These sections generate the HTML code required to visualise the attribute name and value in a browser using a helper function `cimmenu.getAggregateComponentMenu`.

Upon loading CIM data, Pintura iterates through the list of JavaScript objects retrieved from the grid data and executes the static rendering functions of the class associated to each object. For example, when an object of type `ACLineSegment` is found, Pintura calls the static functions defined in the JavaScript class `ACLineSegment` to render the resistance `r` and other attributes specified for this class.

Listing 1: Snippet of JavaScript class file template for CIM

```
class {{class_name}} extends {{sub_class_of}}{
    static attributeHTML(object, cimmenu) {
        let attributeEntries = {{sub_class_of}}.attributeHTML(object, cimmenu);
{{#attributes}}
        if ('cim:{{about}}' in object) {
attributeEntries['cim:{{about}}'] = cimmenu.getAggregateComponentMenu("cim:{{domain}}",
object['pintura:rdfid'], object['{{about}}'], 'cim:{{attributeClass}}' 'cim:{{about}}');
        }
{{/attributes}}
        return attributeEntries;
      }
    ...
};
```

5.2 | CIM++

As part of the CIM++ software environment, the C++ de-/serialiser library libcimpp requires a code generation at two stages. First, the C++ codebase, and second, the unmarshalling code is generated as explained in the following.

5.2.1 | CIM C++ class generation by CIMgen

The CIM C++ codebase provides libcimpp the knowledge of all classes specified in CIM with their associations, aggregations, compositions, and the inheritance hierarchy. It consists of `.hpp` and `.cpp` files with class declarations and con-/destructor definitions. The actual mapping of the CIM ontology to C++ with its Standard Template Library (STL) is presented in [17]. One type of classes defined by the CIM ontology are the so-called *primitive (data) types* such as `String`, `Integer`, `Float`, `Decimal`, and `Boolean` corresponding to intrinsic data types of many programming languages. However, UML (i. e. XMI) and the RDFSs do not define on which intrinsic C++ data types or classes these primitive CIM types must be mapped. Therefore, CIMgen maps these primitive data types to customised CIM++ classes based on intrinsic C++ data types.

List. 2 shows the template file used by CIMgen for the generation of the C++ implementation files of all CIM classes. It contains various Mustache tags for header file includes, the constructor plus destructor, and a class factory function. `{#langPack.create_assign}` stands for a Mustache *lambda* tag (referring to *lambda functions*) which allows to call the `create_assign` function of the according CIMgen module `langPack` for C++. This function generates the assignment functions that are needed for, for example, the attributes of primitive CIM types.

Listing 2: Snippet of C++ class implementation file template for CIM

```
#include <sstream>
#include "{{sub_class_of}}.hpp"
#include "{{class_name}}.hpp"
{{#attributes}}
#include "{{> class}}.hpp"
{{/attributes}}
...
{{class_name}}::{{class_name}}() {};
{{class_name}}::≃{{class_name}}() {};
...
{{#attributes}}
{{#langPack.create_assign}}{{.}}
{{/langPack.create_assign}}
{{/attributes}}
...
void {{class_name}}::addPrimitiveAssignFnsTo
Map(std::unordered_map<std::string,
assign_function>& assign_map) {
{{#attributes}}
    {{> insert_assign}}
{{/attributes}}
}
...
BaseClass* {{class_name}}_factory() {
    return new {{class_name}};
}
...
```

5.2.2 | Unmarshalling in CIM++

Like pure XML, also RDF/XML documents can be read by standard XML parsers. The type utilised by libcimpp is called an event-based Simple API for XML (SAX) parser which traverses XML documents linearly and triggers event callbacks (i.e. a certain type of functions). For instance, a certain callback function is called by the SAX parser in case it encounters characters representing no XML tag. These characters are passed to an *assignment function* which interprets the characters to values and tries assigning them to the right attribute of the respective CIM object. The simplest way for the initialisation of attributes with the values read from RDF/XML document would be using *reflection* capabilities of the programming language which is the ability to examine, introspect, and modify the object's structure and behaviour at the program's runtime. For instance, this would allow the program to iterate through all attributes of an object, investigate their names and types, and assign the concerning values. Contrary to *dynamic* programming languages such as Python, providing reflection and even *object runtime alternation*, C++ as a *static* and ahead-of-time compiled language provides only very limited reflection mechanisms. Therefore, for each class attribute there must be an according assignment function implemented in C++. Since the syntax of each assignment function has the same structure, a template file can be written which is rendered for each attribute of all CIM classes. The same applies to other functions to be implemented for libcimpp. More details on the unmarshalling code can be found in [17].

5.2.3 | Unmarshalling code generation by CIMgen

Contrary to the original unmarshalling code generation of CIM++, the callback (e.g. assignment) functions are not generated separately to the CIM class files. Originally, the CIM C++ classes (i.e. CIM C++ codebase) were generated by a UML editor and the unmarshalling code later on by the unmarshalling generator based on the previously generated CIM C++ codebase. One advantage of the new approach is that all callback functions related to a class are implemented in the appropriate class implementation file. This makes the whole source code more readable and therefore better maintainable. Moreover, now a white list can be specified during the build process

of libcimpp specifying the CIM classes which should be supported. This can reduce libcimpp's compile time and size if support for only a few CIM classes is needed. The evaluation of the new code generation by CIMgen is presented in Section 6.

5.3 | CIMpy

The de-/serialiser library CIMpy was developed in the Python language. It encompasses two core modules: `cimimport` for the unmarshalling of RDF/XML documents and `cimexport` for the marshalling of CIM objects.

5.3.1 | CIM Python class generation by CIMgen

CIMpy is based on the CIM Python class code generated by CIMgen using the template in List. 3. Due to the dynamic typing in Python, the class code generation does not necessitate an explicit mapping of the CIM ontology's primitive data types to language intrinsic data types. Yet, default values are set according to the primitive data types specified by CIM for class attributes by means of the Mustache lambda tag `#setDefault`, where the corresponding function `set_default` is part of CIMgen's `langPack` for Python. Besides, the initialization of parent class attributes is enabled by insertion of a recursive parent class constructor call with `super().__init__`. CIMgen also processes the multiplicity of associations according to the CIM specification. As Python provides reflection and object runtime alternation, no unmarshalling code generation is performed by CIMgen. Instead, the setting and getting of attributes can be handled with the Python's intrinsics `setattr` and `getattr`. In the following, we focus on the explanation of the marshalling in CIMpy.

5.3.2 | Marshalling in CIMpy

CGMES defines profiles based on data model subsets with the purpose of providing interfaces for different types of data exchange. Each instance of a class together with its attributes and associations can be serialised to certain profiles that are defined by the CGMES. Therefore, as it can be seen in List. 3, each class has the two dictionaries `possibleProfileList` and `serializationProfile`, which hold information about the profiles to which the class instance with its attributes and its associations can and will be serialised to.

CIMpy serialises to a set of active profiles that are defined by the user or the software using CIMpy. For each activated profile, CIMpy generates a dedicated RDF/XML document, for which, analogously to CIMgen, CIMpy uses chevron as template engine. List. 4 shows a snippet of the template file for the serialisation of CIMpy objects to RDF/XML. `#set_attributes_or_reference` is again a Mustache lambda tag, which refers to a function that sets attributes or resolves references to other instances.

Listing 4: Snippet of Python serialisation file template for CGMES

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF {{#namespaces}}xmlns:{{key}}
="{{url}}" {{/namespaces}}>
...
{{#classes}}
<cim:{{name}} rdf:ID="{{mRID}}">
{{#attributes}}
       <cim:{{attr_name}}{{#set_attributes_or
_reference}}{{value}}@{{attr_name/
set_attributes_or_reference}}
{{/attributes}}
</cim:{{name}}>
{{/classes}}
...
</rdf:RDF>
```

Listing 3: Snippet of Python class implementation file template for CGMES

```
from {{ClassLocation}} import {{sub_class_of}}
class {{class_name}}({{sub_class_of}}):
    ...
    possibleProfileList = {
        'class': [{{#class_origin}}cgmesProfile.{{origin}}.value, {{/class_origin}}],
        {{#attributes}}'{{label}}': [{{#attr_origin}}cgmesProfile.{{origin}}.value,
        {{/attr_origin}}], {{/attributes}} }
    serializationProfile = {}
    ...
    def __init__(self,
        {{#attributes}}{{label}} ={{#setDefault}}{{dataType}}{{/setDefault}},{{/attributes}}
        {{#super_init}}*args, **kw_args{{/super_init}}):
    {{#super_init}}
    super().__init__(*args, **kw_args)
    {{/super_init}}
    {{#attributes}}
    self.{{label}} = {{label}}
    {{/attributes}}
    ...
```

6 | EVALUATION BY USE CASES

We evaluate the generation of codebases and their integration into the above mentioned software projects by application in use cases which cover the visualisation and simulation of a benchmark grid as well as the change between different versions of CGMES specifications.

6.1 | Topology visualisation with Pintura

List. 5 shows the Pintura JavaScript class `IdentifiedObject`, which derives from `BaseClass` and renders the attributes of an instance of the class. Here, the section for the `shortName` attribute is depicted. Eventually, the function `getAggregateComponentMenu` returns the HTML code to render the attributes and their values in a browser.

Listing 5: Snippet of the rendered JavaScript implementation of the CGMES class IdentifiedObject

```
class IdentifiedObject extends BaseClass {
    static attributeHTML(object, cimmenu) {
    ...
        if ('cim:IdentifiedObject.shortName' in object) {
attributeEntries['cim:IdentifiedObject.shortName'] = cimmenu.getAggregateComponentMenu(
"cim:IdentifiedObject",
object['pintura:rdfid'], object['IdentifiedObject.shortName'],
'cim:String', 'cim:IdentifiedObject.shortName');
        }
        return attributeEntries;
    }
    ...
};
```

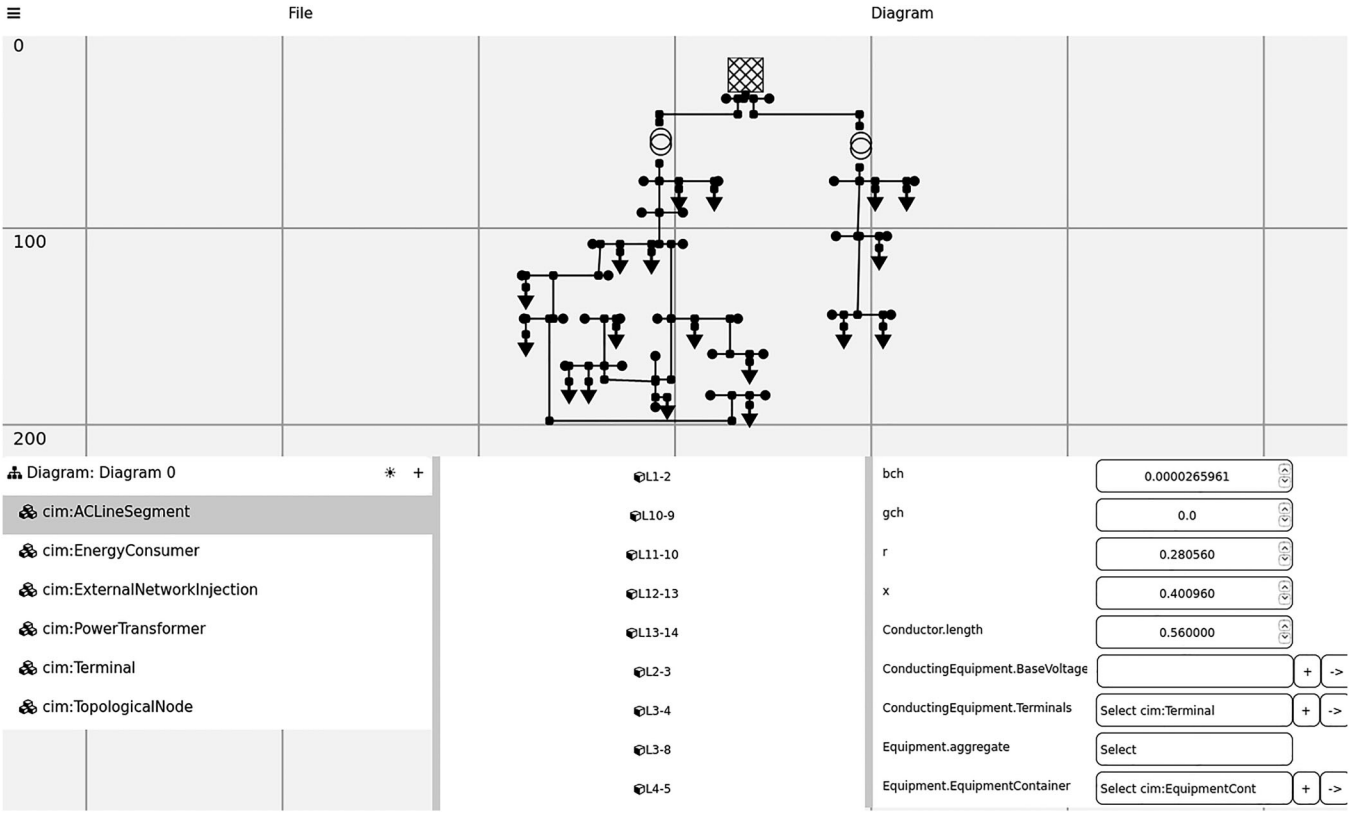

FIGURE 3 Pintura network diagram, object list, and parameters

Figure 3, presents the CIGRE MV benchmark system [38] as an example rendered in Pintura. The screenshot shows the network diagram in the background and several menus in the foreground. In this case, the menus list several instances of `ACLineSegment` and the attribute values of one particular instance, which is selected in the diagram. The RDF/XML documents loaded into Pintura have been generated by the commercial software tool NEPLAN™. This example demonstrates the advantages of a standard format like CGMES regarding interoperability, which enables in this case a proper grid data exchange between the two software tools.

First, we show an example of a CGMES class implementation file in C++ and Python, respectively. In List. 6, the C++ implementation file for the CGMES class `IdentifiedObject` is depicted as it is rendered using the template in List. 2. It shows some generated includes and functions, such as `assign_IdentifiedObject_name`. The latter assigns a string read from the CGMES RDF/XML topology file to the `name` attribute of the `IdentifiedObject` instance.

Listing 6: Snippet of the rendered C++ implementation file for the CGMES class IdentifiedObject

```
#include <sstream>
#include "BaseClass.hpp"
#include "IdentifiedObject.hpp"
#include "DiagramObject.hpp"
...
IdentifiedObject::IdentifiedObject() {};
IdentifiedObject::≃IdentifiedObject() {};
...
bool assign_IdentifiedObject_name(std::stringstream &buffer, BaseClass* BaseClass_ptr1) {
    if(IdentifiedObject* element = dynamic_cast<IdentifiedObject*>(BaseClass_ptr1)) {
        element->name = buffer.str();
        ...
}
...
void IdentifiedObject::addPrimitiveAssignFnsToMap(std::unordered_map<std::string,
assign_function>& assign_map) {
    ...
    assign_map.insert(std::make_pair(std::string("cim:IdentifiedObject.name"),
&assign_IdentifiedObject_name));
}
...
BaseClass* IdentifiedObject_factory() {
    return new IdentifiedObject;
}
...
```

6.2 | Deserialisation and simulation using CIM++ and CIMpy

For an evaluation of the two libraries CIM++ and CIMpy simulations were performed utilising them for a deserialisation of the aforementioned CIGRE MV grid topology data. The RDF schema documents specifying `CGMES_2.4.15_16FEB2016` were used in the following. Here, the processing of CGMES instead of CIM specifications underlines the new capabilities brought by the usage of CIMgen, whereas the original CIM++ codebase toolchain was designed to generate code for libcimpp from CIM UML only.

Such templates also exist for the appropriate class header files generation and for C++ `enum class` constructs (e.g. implementing the CIM / CGMES `UnitMultiplier`) as well as for certain more specialised CIM / CGMES class implementations. More on the code to be generated can be found in [17].

In comparison, List. 7 shows CIMpy's implementation file of `IdentifiedObject`, which was rendered with CIMgen using the Python class template in List. 3 and as well the `CGMES_2.4.15_16FEB2016` specification. It includes an excerpt of the rendered list of possible CGMES profiles for the class instances and the attribute `shortName`, the

TABLE 1 Number of objects instantiated from CGMES C++/Python classes by CIM++ and CIMpy to perform the powerflow simulation of the CIGRE MV benchmark grid using the powerflow solvers of DPsim and pyvolt

CGMES class	Number of objects
ACLineSegment	12
EnergyConsumer	18
ExternalNetworkInjection	1
PowerTransformer	2
PowerTransformerEnd	4
SvPowerFlow	18
SvVoltage	15
Terminal	47
TopologicalNode	15

class constructor, and the initialisation of class attributes. As `IdentifiedObject` inherits from `Base` only, where `Base` does not require the initialisation of any attributes, no code is generated for the template sections marked by the `#super_init` tag in List. 3.

objects were essential for running the powerflow simulation of the CIGRE MV benchmark grid. The corresponding number of these objects is listed in Tab. 1 according to the CGMES C++/Python classes they have been instantiated from.

With both powerflow solvers, which used the deserialisation capabilities of CIM++ and CIMpy, we obtained the same simulation results as with the commercial simulation tool NEPLAN™. This shows that the deserialisation of CIM++ and CIMpy leads to a system model representation in both languages that properly reflects the involved grid components together with their parametrisation and interconnections in the performed powerflow simulations.

As deserialiser, CIMpy has already been employed as well for grid monitoring and control service implementations within the European project Service Oriented Grid For The Network of The Future (SOGNO) [42]. It should be pointed out that these evaluations by use case are limited to the validation of a subset of CGMES. For a comprehensive and automated validation of the generated codebases, which verifies the consistency with the complete CGMES, a corresponding methodology needs to be developed, which however is beyond the scope of this paper.

Listing 7: Snippet of the rendered Python implementation for the CGMES class IdentifiedObject

```
from cimpy.cgmes_v2_4_15.Base import Base
class IdentifiedObject(Base):
    ...
    possibleProfileList = {'class':[cgmesProfile.DI.value, cgmesProfile.DY.value,
cgmesProfile.EQ.value, ...], ...,
'shortName': [cgmesProfile.EQ.value, cgmesProfile.TP.value, ] }
    serializationProfile = {}
    def __init__(self, DiagramObjects = "list", mRID = '', name = '', description = '',
energyIdentCodeEic = '', shortName = '',  ):
        self.DiagramObjects = DiagramObjects
        self.mRID = mRID
        self.name = name
        self.description = description
        self.energyIdentCodeEic = energyIdentCodeEic
        self.shortName = shortName
```

Using the generated classes, we run powerflow simulations of the CIGRE MV benchmark grid, as visualised in Figure 3, in both languages C++ and Python. For C++, the powerflow solver of the real-time simulator DPsim [39] [40] was applied together with the use of CIM++ for deserialising the grid's RDF/XML documents. In case of Python, the powerflow solver of the package pyvolt [41] was used with CIMpy as deserialiser.

Both deserialiser libraries, CIM++ and CIMpy, were processing the grid data according to the aforementioned CGMES specification. The data of the CIGRE MV benchmark grid consisted of the three CGMES profiles `EQ`, `TP` and `SV` stored as RDF/XML documents. The three documents included in total the description of 300 CGMES objects. Only a subset of these

6.3 | Code generation from further CGMES versions

In addition, we have considered CIMgen's capability to change readily to codebases that are compliant with further CGMES versions. In the previous section, CIMgen's generated code for unmarshalling with CIM++ was based on `CGMES_2.4.15_16FEB2016`. The code generation was executed for two further CGMES versions: `CGMES_2.4.13_18DEC2013` and `CGMES_2.4.15_27JAN2020`. The generated codebases were included in CIM++. They were verified successfully by obtaining the same simulation results with the DPsim simulator as in the previous section.

7 | CONCLUSION AND OUTLOOK

This paper presents an automated mapping from CIM / CGMES specifications to language specific source code. The approach is based on XMI and RDF schema documents as well as templates written in the Mustache template language and was implemented as the software project CIMgen. Contrary to former approaches, CIMgen does not rely neither on any meta models for the intermediate representations of CIM / CGMES nor on any further intermediate software tools for the code generation. Therefore, CIMgen only depends on a common XML handling library (xml2dict) and a template engine (chevron). This and the choice of Python as a modern scripting language intend to make CIMgen easily maintainable and flexible extendable for other software projects based on CIM and CGMES. The applicability of the approach and its implementation is shown on the one hand with code generations for the de-/serialiser libraries CIM++ and CIMpy, which are both integrated into power grid simulators and tested in benchmark system simulations. On the other hand, CIMgen is applied for the development of a graphical CIM / CGMES editor, the web based application Pintura. With CIMgen the mentioned and other CIM / CGMES based software projects programmed in arbitrary programming languages can be kept up-to-date with minimum programming effort as demonstrated with CIM++ covering various CIM / CGMES versions. Given that, CIMgen leverages an easy CIM / CGMES based data exchange between multiple entities, for example, within the European project Artificial Intelligence for Next Generation Energy (I-NERGY) [43]. As open-source software project, CIMgen can be investigated, improved, and extended by anyone for the own purpose.

AUTHOR CONTRIBUTIONS

Jan Dinkelbach: Conceptualization, software, writing - original draft, writing - review and editing. Lukas Razik: Conceptualization, software, writing - original draft, writing - review and editing. Markus Mirz: Conceptualization, software, writing - original draft, writing - review and editing. Andrea Benigni: Funding acquisition, supervision. Antonello Monti: Funding acquisition, supervision.

ACKNOWLEDGEMENTS

The work of J.D., M.M. and A.M. was supported by SOGNO and I-NERGY, which are European projects funded from the European Unions Horizon 2020 programme under Grant Agreement Nos. 774613 and 101016508, respectively. The work of L.R. and A.B. was supported by the Helmholtz Association under the Joint Initiative Energy Systems Integration.

CONFLICT OF INTEREST

The authors have declared no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Jan Dinkelbach https://orcid.org/0000-0003-4996-0697
Lukas Razik https://orcid.org/0000-0002-0820-2489
Markus Mirz https://orcid.org/0000-0003-3209-9861
Andrea Benigni https://orcid.org/0000-0002-2475-7003
Antonello Monti https://orcid.org/0000-0003-1914-9801

REFERENCES

1. Buchholz, B.M., Styczynski, Z.A.: Smart Grids: Fundamentals and Technologies in Electric Power Systems of the future. Springer, Berlin, Heidelberg (2020)
2. Kumari, A., Tanwar, S., Tyagi, S., Kumar, N., Obaidat, M.S., Rodrigues, J.J.P.C.: Fog computing for smart grid systems in the 5G environment: challenges and solutions. IEEE Wireless Commun. 26(3), 47–53 (2019)
3. Garau, M., Anedda, M., Desogus, C., Ghiani, E., Murroni, M., Celli, G.: A 5G cellular technology for distributed monitoring and control in smart grid. In: 2017 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), pp. 1–6. IEEE, Piscataway (2017)
4. Liu, Y., Wang, J., Li, J., Niu, S., Song, H.: Machine learning for the detection and identification of internet of things devices: A survey. IEEE Internet Things J. 9(1), 298–320 (2022)
5. Mercurio, A., Di.Giorgio, A., Cioci, P.: Open-source implementation of monitoring and controlling services for EMS/SCADA systems by means of web services- IEC 61850 and IEC 61970 standards. IEEE Trans. Power Delivery 24(3), 1148–1153 (2009)
6. Pau, M., Mirz, M., Dinkelbach, J., Mckeever, P., Ponci, F., Monti, A.: A service oriented architecture for the digitalization and automation of distribution grids. IEEE Access 10, 37050–37063 (2022)
7. CIM Users Group: CIMug. Accessed 15 Sept 2022. http://cimug.ucaiug.org
8. IEC: IEC 61970-301:2012 Energy management system application program interface (EMS-API) – Part 301: Common Information Model (CIM) base. IEC, Stevenage (2012)
9. IEC: IEC 61968-11:2013 Application Integration at Electric Utilities - System Interfaces for Distribution Management – Part 11: Common Information Model (CIM) Extensions for Distribution. IEC, Stevenage (2012)
10. IEC: IEC 62325-301:2014 Framework for Energy Market Communications – Part 301: Common Information Model (CIM) Extensions for Markets. IEC, Stevenage (2014)
11. Lefebvre, T., Englert, H.: IEC TC57 Power system management and associated information exchange. Accessed 22 Feb 2020. https://www.iec.ch/resources/tcdash/Poster_IEC_TC57.pdf
12. Uslar, M., Specht, M., Rohjans, S., Trefke, J., González, J.M.: The common information model CIM: IEC 61968/61970 and 62325 – A practical introduction to the CIM. Springer, Berlin, Heidelberg (2012)
13. Mirz, M., Razik, L., Dinkelbach, J., Tokel, H.A., Alirezaei, G., Mathar, R., et al.: A cosimulation architecture for power system, communication, and market in the smart grid. Complexity 2018, 1–12 (2018)
14. IEC.: IEC TS 61970-600-1: Common Grid Model Exchange Specification (CGMES) – Structure and rules. (2017). Accessed 20 May 2020. https://webstore.iec.ch/preview/info_iec61970-600-1%7Bed1.0%7Den.pdf
15. ENTSO-E.: Common Grid Model Exchange Standard (CGMES) Library. Accessed 20 May 2020. https://www.entsoe.eu/digital/cim/cim-for-grid-models-exchange/
16. Wanstrath, C.: Mustache manual (2009). Accessed on 24 May 2020. https://mustache.github.io/mustache.5.html
17. Razik, L., Mirz, M., Knibbe, D., Lankes, S., Monti, A.: Automated deserializer generation from CIM ontologies: CIM++—an easy-to-use and automated adaptable open-source library for object deserialization in C++ from documents based on user-specified UML models following the Common Information Model (CIM) standards for the energy sector. Comp. Sci. - Res. Develop. 33(1), 93–103 (2018)
18. Agarwal, B.B., Tayal, S.P., Gupta, M.: Software Engineering and Testing. Computer Science Series. Jones & Bartlett Learning, Sudbury, MA (2010)

19. Razik, L., Dinkelbach, J., Mirz, M., Monti, A.: CIMverter—a template-based flexibly extensible open-source converter from CIM to Modelica. Energy Informatics 1(1), 47 (2018)
20. CIMgen Developers: CIMgen. https://www.fein-aachen.org/projects/cimgen/
21. Lincoln, R.: GitHub - rwl/PyCIM: Python implementation of the Common Information Model. Accessed 20 May 2020. https://github.com/rwl/PyCIM
22. Lincoln, R.: GitHub - Code generation flow - Issue #26 - rwl/PyCIM. Accessed 20 May 2020. https://github.com/rwl/PyCIM/issues/26
23. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd ed. Pearson Higher Education, Reading, MA (2004)
24. Quirolgico, S., Assis, P., Westerinen, A., Baskey, M., Stokes, E.: Toward a formal common information model ontology. In: Web Information Systems – WISE 2004 Workshops, pp. 11–21. Springer, Berlin, Heidelberg (2004)
25. McMorran, A.W.: An introduction to IEC 61970-301 & 61968-11: The common information model. Univ. Strathclyde 93, 124 (2007)
26. Bray, T., Paoli, J., Sperberg.McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML). World Wide Web J. 2(4), 27–66 (1997)
27. W3C.: XML Schema Definition Language (XSD) 1.1 Part 1: Structures (2012). Accessed 20 May 2020. https://www.w3.org/TR/xmlschema11-1/
28. Pan, J.Z.: Resource description framework. In: Handbook on Ontologies, pp. 71–90. Springer, Berlin, Heidelberg (2009)
29. W3C: RDF 1.1 XML Syntax (2014). Accessed 20 May 2020. https://www.w3.org/TR/rdf-syntax-grammar/
30. W3C: RDF Schema 1.1 (2014). Accessed 20 May 2020. https://www.w3.org/TR/rdf-schema/
31. w3schools: XML RDF. Accessed 20 May 2020. https://www.w3schools.com/xml/xml_rdf.asp
32. OMG: XML Metadata Interchange (XMI) Specification. Accessed 20 May 2020. https://www.omg.org/spec/XMI/2.5.1/PDF/changebar
33. Chevron Developers: GitHub - noahmorrison/chevron: A Python implementation of mustache. Accessed 20 May 2020. https://github.com/noahmorrison/chevron
34. Pintura Developers: Pintura - Graphical CIM Editor. Accessed 24 Aug 2021. https://www.fein-aachen.org/projects/pintura/
35. Monti, A., Stevic, M., Vogel, S., De-Doncker, R.W., Bompard, E., Estebsari, A., et al.: A global real-time superlab: enabling high penetration of power electronics in the electric grid. IEEE Power Electron. Mag. 5(3), 35–44 (2018)
36. CIM++ Developers: CIM++ - CIM for C++. Accessed 24 Aug 2021. https://www.fein-aachen.org/projects/cimpp/
37. CIMpy Developers: CIMpy - Python package for CIM data. Accessed 24 Aug 2021. https://www.fein-aachen.org/projects/cimpy/
38. Strunz, K., Abbasi, E., Fletcher, R., Hatziargyriou, N., Iravani, R., Joos, G.: TF C6.04.02 : TB 575 – Benchmark Systems for Network Integration of Renewable and Distributed Energy Resources (2014)
39. Mirz, M., Dinkelbach, J., Monti, A.: DPsim - Advancements in power electronics modelling using shifted frequency analysis and in real-time simulation capability by parallelization. Energies 13(15), 3879 (2020)
40. DPsim Developers: DPsim: A Real-Time Power System Simulator. Accessed 24 Aug 2021. https://dpsim.fein-aachen.org
41. Pyvolt Developers: Pyvolt. Accessed 24 Aug 2021. https://github.com/sogno-platform/pyvolt
42. SOGNO Project Partners: SOGNO—D4.3 Report on development, implementation and component tests (2020)
43. I-NERGY Project Partners: I-NERGY—D4.1 I-NERGY Analytics Applications (1st technology release) (2021)

How to cite this article: Dinkelbach, J., Razik, L., Mirz, M., Benigni, A., Monti, A.: Template-based generation of programming language specific code for smart grid modelling compliant with CIM and CGMES. J. Eng. 2023, 1–13 (2022). https://doi.org/10.1049/tje2.12208