

# **Implementation Profile:**

## ***Modeling Environment***

### **Development of an Energy Services Interface for the EGoT**

#### **WORK PERFORMED UNDER AGREEMENT**

DE-OE0000922

Portland State University  
1900 SW 4<sup>th</sup> Ave  
Portland, OR 97201

**Period of Performance:** 7/13/2020 to 9/30/2023

**Submitted:** January 4, 2024

**Revision:** 1.0

#### **PRINCIPAL INVESTIGATOR**

Robert Bass, Ph.D.  
503-725-3806  
robert.bass@pdx.edu

#### **BUSINESS CONTACT**

Patti Fylling  
503-725-6584  
spa\_mcecs@pdx.edu

#### **SUBMITTED TO**

U. S. Department of Energy  
National Energy Technology Laboratory  
DOE Project Officer: Mario Sciulli

**This report does not contain any proprietary, business sensitive, or other information not subject to public release.**

## Implementation Profile: *Modeling Environment*

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000922.

**Disclaimer:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

## Revision History

Date	Version	Description	Author
6/1/21	1.0	Document creation	R. Bass
7/9/21	1.1	First draft written	S. Keene
8/13	1.2	ME architecture	S. Keene
8/18/21	1.3	Fully rewritten for class-based architecture	S. Keene, R.Bass
8/23/21	1.4	Revision of MC implementation	S. Keene, R.Bass
8/26/21	1.5	Addition of product requirements	S. Keene, R.Bass
8/26/21	2.0	Finalize draft. Sent to PNNL for review.	R.Bass
9/13/21	2.1	Revisions from partner reviews and test plan development	R. Bass, S. Keene
9/14/21	2.2	Final Version	R. Bass, S. Keene
3/21/22	3.0	Document review and revision. New terms definitions. Updated Ch 3 text & figures. Updated class descriptions, figures in App A.	S. Keene
3/28/22	3.1	Revision review	R. Bass
3/30/22	3.2	Added App B.3 "DER-S Design," formatting in Apps A & B, new Figure 2.2	S. Keene
4/6/22	3.3	Revision review	R. Bass
4/27/22	3.4	Document review	S. Poudel
5/12/22	3.5	Revision updates. Added numerous figures and tables. Revision review.	S. Keene, R. Bass
6/23/22	3.6	Updated definitions, moved Acronyms section	R. Bass
8/14/23	3.7	Update Figure 2.1, System integration	R. Bass
12/1/23	3.8	Updated GOSensor intro, Table 3.1 headers and caption, and Automatic GO decision mode.	M. Adham
12/3/23	3.9	Updated Automatic GO Decision making process	M. Adham
12/4/23	4.0	Reviewing Automatic GO Decision making process	M. Adham
12/6/23	4.1	Updated the GOTopologyProcessor section and Figure 3.10	M. Adham
12/8/23	4.2	Added App. B.2 "PSU Feeder Simulation example". updated App. B references.	M. Adham
12/10/23	4.3	Updated RWHDEERS class identifiers and EDMCore attributes.	M. Adham
12/15/23	5.0	Final Review	R. Bass

# Implementation Profile:

## *Modeling Environment*

<b>Acronyms</b>	<b>4</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Design Principles	6
1.2 Participants & Definitions	6
<b>2 Modeling Environment Architecture</b>	<b>10</b>
2.1 GridAPPS-D	10
2.2 Grid Models and the Electrical Distribution Model	11
2.3 Model Controller Script	12
<b>3 Implementation</b>	<b>16</b>
3.1 Electrical Distribution Model Implementation	16
3.2 Model Controller Implementation	18
3.3 DER-S Implementation	30
3.4 Grid Operator Implementation	33
<b>Appendix A</b>	<b>37</b>
A.1 MC Callback Classes	37
A.2 MC Input Classes	45
A.3 MC Output Classes	52
<b>Appendix B</b>	<b>60</b>
B.1 Example Simulation Procedure	60
B.2 PSU IEEE 13-Node Feeder Simulation Example	66
B.3 Simulation Process Summary	69
B.4 DER-S Design	74
<b>Appendix C Product Requirements</b>	<b>80</b>
C.1 ME Product Requirements	80
C.2 DER-S Product Requirements	80
C.3 GO Product Requirements	81
C.4 EDM Product Requirements	81
C.5 MC Product Requirements	82

# Acronyms

<b>API</b>	Application Programming Interface
<b>BIS</b>	Battery-Inverter Systems
<b>CIM</b>	Common Information Model
<b>DCM</b>	Distributed Control Module
<b>DER(-S, -EM)</b>	Distributed Energy Resource (Simulated, Electrical Model)
<b>DERMS</b>	DER Management System
<b>EDM</b>	Electrical Distribution Model
<b>EGoT</b>	Energy Grid of Things
<b>GLM</b>	GridLAB-D model file type
<b>GO</b>	Grid Operator
<b>GSP</b>	Grid Service Provider
<b>MC</b>	Model Controller
<b>ME</b>	Modeling Environment
<b>mRID</b>	Master Resource Identifier
<b>SPC</b>	Service Provisioning Customer
<b>TE</b>	Test Engineer
<b>TLS</b>	Transport Layer Security
<b>PNNL</b>	Pacific Northwest National Laboratories
<b>UML</b>	Unified Modeling Language

# 1 Introduction

This implementation profile provides the scope, background, and requirements necessary to implement a Modeling Environment (ME) to test a Distributed Energy Resource (DER) Management System (DERMS). A DERMS is used by an aggregator to dispatch large numbers of DERs in order to provide grid services to a Grid Operator (GO). The ME addresses scalability issues inherent to Hardware-in-the-Loop DERMS simulation; a large number of assets are needed in order to observe effects on the grid from deployment and dispatch of DERs. However, it would be prohibitive to physically procure and install these assets, so it is desirable to have a simulation environment that can model interactions between a DERMS and a mass of simulated DERs within an Electrical Distribution Model (EDM) while also being able to interact with a limited number of physical DERs.

Within the ME, DERs are abstracted into generic models of simulated DERs (DER-S) and electrical model DERs (DER-EM). DER-EMs are the electrical representations of DERs within the grid model, while DER-Ss handle the inputs from physical DERs or data from DER simulations and convert them into electrical data required by DER-EMs. This removes the need to operate a large number of physical DERs. An abstract DER-S representation allows for new types of DER models to be developed and simulated within the ME without major modifications to the grid model or modeling system apart from development of a single new API handled by a DER-S class. The ME allows real time and historical DER data to be used as DER-S inputs, calculates DER-EM grid states via an internal simulator, and provides a configurable simulation of a GO to communicate with external DERMS. Together, these operations provide a full feedback loop between the DERMS asset dispatch and the GO's recognition of how the dispatch affects the grid. The key component of the ME is the Electrical Distribution Model (EDM) which provides an interactable grid model and simulation environment by leveraging the GridAPPS-D app development platform.

The ME was designed as a co-simulation environment for the Portland State University Power Lab's Energy Grid of Things (EGoT) DERMS prototype. However, the abstracted DER-S and GO allow the ME to be used to test other service-based DERMS with minimal effort. As such, while the EGoT prototype will be used for demonstration and example purposes, the more generic term "DERMS" will be used to describe the system being tested by the ME.

## 1.1 Design Principles

The following principles guide the design and development of the ME IP.

1. Develop an open-source, object-oriented system expressly designed to be used, added to, or upgraded by individuals or entities beyond the original designer.
2. Make the system configurable, scalable, and suitable for a variety of tests for a number of grid scales, DER counts, or grid services.
3. Assume DERs will rapidly evolve in type and complexity in the short term, and design the system to be extensible to new or updated DER models.
4. Output data will be in a common, non-proprietary format and without excessive processing; that is, do not develop the system to perform data science, but to generate data suitable for external analysis.
5. Design the system with generalized, easily modifiable modules to allow testing of a wide variety of DERMSs without protocol limitations.

## 1.2 Participants & Definitions

Tables 1.2 through 1.4 list the main participants that interact through information exchange to perform DERMS testing, feedback, and data logging. The number of participants varies depending on the use case.

Participants are classified as *Actors*, which are persons or other external systems; *Collaborative Objects*, which include interacting components other than Actors; and *Products*, which are the Collaborative Objects developed for this DOE-sponsored project.

**Table 1.2** Actors relevant to the Modeling Environment.

Name	Type
Grid Operator (simulated)	application
Grid Services Provider (DERMS operator)	organization
Test Engineer	person

**Table 1.3** Collaborative objects relevant to the Modeling Environment.

Name	Type
Distributed Control Module	agent
Distributed Energy Resource	device
Distributed Energy Resource - Simulated (DER-S)	device or application
Electrical Distribution Model	application
Model Controller	application

**Table 1.4** Products relevant to the Modeling Environment.

Name	Type
EGoT Server/Client System	application
Modeling Environment	application

### 1.2.1 Functionalities and Responsibilities

This subsection presents the functionalities and responsibilities of the system actors and products that pertain to the Modeling Environment as well as the EGoT System to which it is applied.

#### 1.2.1.1 Actors

**Table 1.5** Grid Operator (GO)

<b>Functionality</b>	A GO seeks grid services from GSPs in order to achieve operational objectives, which are 1) maintaining operations within the physical constraints that must be honored in order to prevent damage to grid components and equipment, or 2) operational goals associated with stable, reliable, and economical delivery of power at nominal conditions.
<b>Responsibilities</b>	<ul style="list-style-type: none"><li>• Engage with GSPs to acquire grid services to achieve operational objectives.</li><li>• Design and fund incentive programs to attract GSP and/or SPC participation to implement operational objectives.</li><li>• Provide DER topological assignment information during the registration process</li></ul>

**Table 1.6** Grid Service Provider (GSP)

<b>Functionality</b>	A GSP provides grid services to a GO through the dispatch of DER that have subscribed to a GO program. Aggregation and dispatch are achieved using a DERMS. Grid services are the means by which a GO achieves operational objectives.
<b>Responsibilities</b>	<ul style="list-style-type: none"><li>• Provide grid services to GOs.</li><li>• Evaluate its aggregation of DER assets to determine a menu of grid services to offer to GOs, prioritized based on the priority operational objectives of GOs.</li><li>• Entice SPCs to subscribe to DER aggregation programs</li><li>• Exchange information according to the EGoT Server/client Implementation Profile.</li></ul>

**Table 1.7** Test Engineer (TE)

<b>Functionality</b>	The TE is a person or group of people who configure or operate the modeling environment.
<b>Responsibilities</b>	<ul style="list-style-type: none"> <li>• Develop electrical grid models and enter into ME database</li> <li>• Plan and design tests</li> <li>• Configure MC with test parameters</li> <li>• Implement and configure DER-S</li> <li>• Operate ME to perform tests</li> <li>• Collect and analyze measurement logs</li> <li>• Maintain and troubleshoot ME system</li> </ul>

### 1.2.1.2 Products

**Table 1.8** EGoT Server/Client System

<b>Functionality</b>	The EGoT server and client facilitate TLS and HTTP communications using the IEEE 2030.5 resource models. The client and server are also responsible for translating the common IEEE 2030.5 models into the specific DER and GO interfaces to implement controls and energy services.
<b>Responsibilities</b>	<ul style="list-style-type: none"> <li>• Authenticate client/server</li> <li>• Encrypt/Decrypt HTTP communications</li> <li>• Validate IEEE 2030.5 resource models using xml schema</li> <li>• Update resources based on polling rates, event status, or pub/sub</li> <li>• The client interfaces with DER using the <i>flow reservation</i> resources and <i>DER</i> function sets of IEEE Std 2030.5.</li> <li>• The server interfaces with a GO to provide Grid-DER services.</li> </ul>

**Table 1.9** Modeling Environment

<b>Functionality</b>	The ME is a configurable power grid modeling and simulation system, which is capable of modeling DERs and their effect on the grid based on real time or historical input. These simulations can produce both CSV log outputs for analysis as well as real-time response via a simulated GO; this allows for GSP-in-the-loop simulation.
<b>Responsibilities</b>	<ul style="list-style-type: none"> <li>• EDM simulates the power grid over a selected period of time.</li> <li>• Provides a database of grid models to simulate; new models can be developed and added by TE</li> <li>• Grid states measured within grid model and provided as output to logs or external systems</li> <li>• DER modeled as generic electrical loads/sources within EDM</li> <li>• External scripts or devices emulate the operation of DER and provide operating data to the ME as inputs to generic DER loads</li> <li>• Provides timekeeping function to synchronize external devices/programs with simulation, for functional testing</li> </ul>

## 1.2.2 Definitions

Below are definitions of all terms required to properly interpret this document, as defined for this project.

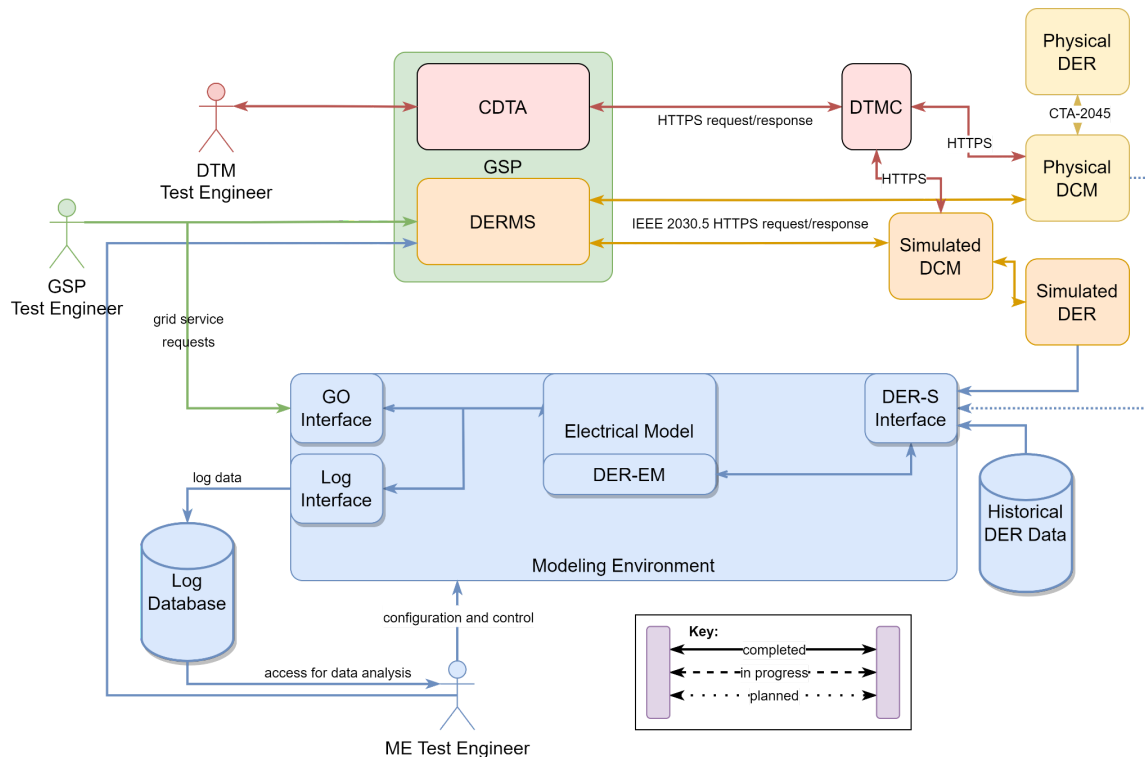
**Table 1.10** Modeling Environment Collaborative Object Definitions

	Definition
Distributed Control Module	A DCM is a client that requests resources from a DERMS server. It provides gateway service between communications protocols used by the DERMS and communications protocols used by DER. It serves as a user-agent on behalf of the SPC to autonomously make resource service request decisions.
Distributed Energy Resource	DERs are customer-owned generation, storage, and load assets that are grid-enabled. These resources are located behind a customer meter.
Distributed Energy Resources - Electrical Model	The electrical model representation of a DER or DER-S within the EDM that models the electrical characteristics of the DER.
Distributed Energy Resources - Simulated	A system, application, or script that simulates the parameters and operations of a physical DER and outputs the resultant electrical effects.
Electrical Distribution Model	An electric model of a distribution system used to determine the electrical impacts that a grid service dispatch would cause.
Model Controller	An application that manages operation of the ME simulations and interfaces.
Modeling Environment	A test environment for simulating the interactions between the GO, GSPs, and SPCs actors within the EGoT prototype system. The ME includes an EDM, and provides points of interface between the EDM and actors within the EGoT.
Master Resource Identifier	An mRID is an alphanumeric code used by GridAPPS-D to reference objects, systems, measurements or controls within a grid model. Not human readable.
Unique Identifier	Unique information differentiating a single DER input from others. Used to associate DER input information with DER-EMs by associating a single unique identifier with a control mRID for a DER-EM. Unique identifiers are alphanumeric strings provided by DER inputs or generated by a DER-S and are likely to be human readable.
Locational Identifier	An alphanumeric string referring a DER input to a location in the grid model. If topological processing is not in use, this will be the bus to which the DER should be assigned; if topological processing is used, this will be an alphanumeric string that identifies each layer of the topology to which the DER is connected.
DER input	DER operating data that is provided to a DER-S to be parsed and converted into a format usable by the simulation. May include one or more representations of DERs, each of which must have a unique identifier and a locational identifier.

## 2 Modeling Environment Architecture

The ME is an electrical grid simulation platform designed to support testing of a DERMS and provide a means for analyzing effects of DER dispatch on the electrical grid. We have selected GridAPPS-D as the EDM platform, which provides means for modeling distribution systems and includes a grid states solver. A MC coordinates simulations and provides input and output capabilities. Figure 2.1 shows the components of the ME and its relationships with other parts of the EGoT project.

The MC provides communications and processing between the EDM, inputs, and outputs. DER-Ss provide input data to the DER-EM; these input data may come from a model, an historical data archive, or a hardware interface such as a DCM. Outputs from the EDM come in two forms: time-coded logs as well as real-time grid state data. These are available to a simulated GO, which in turn feeds back information to the GSP's DERMS, including grid service requests and locational data.



**Figure 2.1** The Modeling Environment flowchart. Objects and lines in blue fall within the scope of the ME product. The ME interacts with other project objects, the GSP (green), the DTM System (red), the DERMS (orange), and physical DER (yellow).

### 2.1 GridAPPS-D

The ME is built using the GridAPPS-D platform. GridAPPS-D is an open-source application development platform designed by the Pacific Northwest National Laboratory to provide an architecture by which applications can easily communicate with grid

simulation programs. The network measurement data in GridAPPS-D originates from a three-phase unbalanced distributor simulator.

The GridAPPS-D installation, and by extension the EDM, is located within a Docker container. The remainder of the MC is contained in a script external to the GridAPPS-D simulation; however, the MC and any other potential application can be packaged within the GridAPPS-D container for deployment purposes. GridAPPS-D provides a Python API library that facilitates communication between the script and GridAPPS-D via discrete “topics”, which are communications channels designed for a specific purpose such as simulation input messages or database queries.

GridAPPS-D applications interface with a Blazegraph database, which is also included with GridAPPS-D within the Docker container. This database includes grid models in the Common Information Model (CIM) format. CIM contains identification numbers for all components and measurements within this model, and these identifiers (or mRIDs) are used by the MC to direct inputs to the proper DERs or identify measurement points, for example.

During a typical GridAPPS-D simulation, the Test Engineer (TE) selects a model from the database (such as, for example, the IEEE 13-node test feeder, which is included in the package) as well as a simulation start time, duration, and other relevant options. Upon execution, GridAPPS-D converts the CIM model to a GridLAB-D model and performs the simulation in real time while awaiting messages on the platform topics. GridAPPS-D also provides the ability to send regular timekeeping and measurement messages to the MC via callback objects, discussed below in Section 3.2).

## ***2.2 Grid Models and the Electrical Distribution Model***

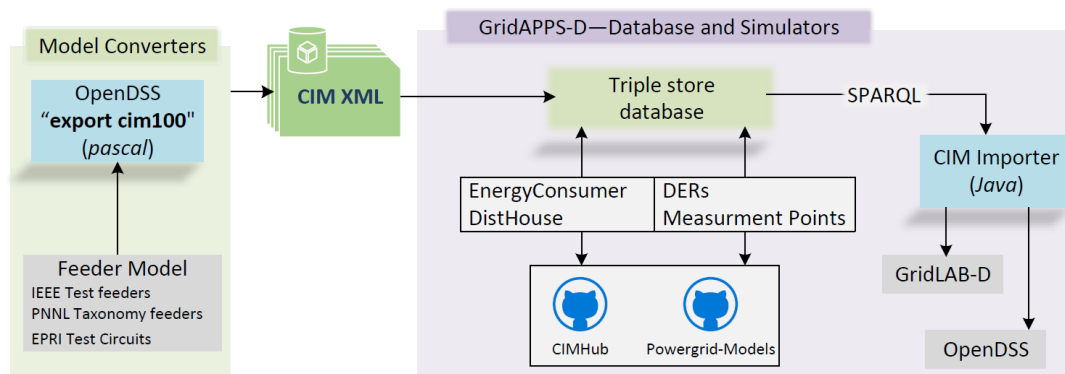
The Blazegraph database contains several built-in grid models, such as IEEE feeders, PNNL taxonomy feeders, and OpenDSS/EPRI circuits in the CIM XML format. These models can be run within the simulation directly. However, these models do not contain the DER-EM representations needed to test input to the simulation. Existing grid models modified to contain DER-EMs are required for EGoT testing. These DER-EMs should be generalizable to a variety of DER types to ensure the ME is extensible as new DERs are developed, and their usefulness explored via simulation.

CIM-based Battery Inverter System (BIS) models can serve as DER-EMs within the grid model. They can be easily controlled via the MC DER input functions, and can be operated to model a wide variety of DER loads. Furthermore, CIMHub<sup>1</sup> allow BIS models to be added directly to an existing model in the database at required locations without going through the process of generating, converting, and ingesting a new model.

New grid models can be created as well. They must be in the CIM format to be ingested into the database; other formats such as GridLAB-D files need to be converted to CIM before ingestion. The BIS models may be added during model creation or using the script, as above.

---

<sup>1</sup> <https://github.com/GRIDAPPSD/CIMHub>, see App. B.1.2



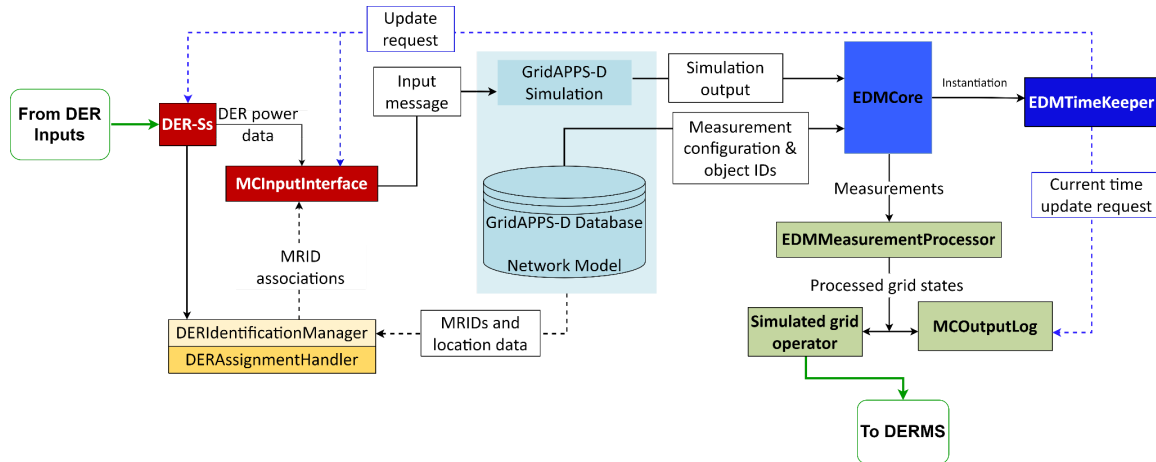
**Figure 2.2** A diagram of the model ingestion process. Grid models generated in a variety of file formats can be converted to CIM XML by OpenDSS, which can be added to GridAPPS-D's database which contains several IEEE Test Feeders by default. Modifications to the models can be made using the CIMHub and Powergrid-Models scripts, such as the addition of Battery Inverter System models to be used as DER-EMs (see appendix B.1.3). Then, during the simulation run process, the model is taken from the database and converted to the GLM format for use by the simulation (as well as providing an OpenDSS file for validation purposes.)<sup>2</sup>

## 2.3 Model Controller Script

The MC is an object-oriented Python script that controls the simulation and provides interfaces for input and output, Figure 2.2. The class-based structure of the script allows the MC to be divided into several 'actors' responsible for individual tasks, such as handling input messages to the EDM, or providing an interface between the MC and the GSP.

GridAPPS-D assists in this implementation by providing "callback methods" that can be included in user-written classes, which then become "callback classes". These callback methods are automatically called based on inputs from GridAPPS-D. For example: the EDMTimekeeper's callback method is called every time a log message is received from GridAPPS-D. These log messages include a wide variety of data, including incrementation messages tied to the internal simulator's time keeping function; these messages are parsed and whenever an incrementation is detected, the EDMTimekeeper can then call a method to perform once-per-timestep functions for the MC. The EDMMeasurementProcessor functions similarly, receiving messages containing grid state measurements as they're delivered by the GridAPPS-D simulation. This occurs once every three seconds. Most MC functions are timed by the EDMTimekeeper's "simulation timestep" of one second, but measurement processing is timed by the "measurement timestep" of three seconds.

<sup>2</sup> S. Poudel et al, "[Modeling Environment for Testing a Distributed Energy Resource Management System \(DERMS\) using GridAPPS-D Platform](#)," IEEE Access, 10:77383-77395, 2022.



**Figure 2.3** An overview of the Class-based architecture of the Model Controller.

### 2.3.1 Electrical Distribution Model Classes

The callback classes that make up the core of the EDM-MC communication pipeline are:

- **EDMCore** - Manages configuration and startup of the script as well as instantiation of all of the necessary classes.
- **EDMTimekeeper** - Called once per second, this keeps track of the simulation start time and current time, and also sends signals to all objects that need to be updated once per second (such as the MCInputInterface).
- **EDMMeasurementProcessor** - Once every measurement timestep (three seconds) this object receives updated grid states from the EDM, which it then organizes, appends necessary locational data, and sends to the MCOOutputLog and GOSensor objects.

Two special cases are the **MCInputInterface** and the **MCConfiguration** classes. The MCInputInterface class manages messages sent to the EDM input topic. This class does not contain a callback method; however, it does interface directly with the EDM. In this case it provides inputs to the EDM rather than receiving outputs, and these inputs are delivered once per timestep on prompting by the EDMTimekeeper. The MCConfiguration attributes should be customized by the user as they contain file directories, paths, the list of active DER-Ss for the simulation, and any other global configuration data necessary for the script to properly utilize inputs and generate outputs.

### 2.3.2 Distributed Energy Resource Classes

The DERs in the electrical model (DER-EMs) are generically represented as battery-inverter system (BIS) models. BIS models provide a wide variety of sourcing, sinking, and ramp rate controls, which can be adjusted to represent a wide variety of DERs. The control inputs to DER-EMs come from simulated DERs (DER-S). Each DER-S may represent one or multiple DERs and handles input and processing from external simulations, hardware, or data. These DER-Ss provide electrical operating data as an output. DER-Ss are assigned to DER-EMs at the proper topological location at the start

of the simulation. Data associating the DERs to one another is held in a table for use by the input processor and output classes.

Each DER-EM within the model has a unique control mRID number, which provides a location for inputs to be delivered via the appropriate GridAPPS-D messaging function. Its location within the grid model can also be queried. However, the emulators or hardware that provide inputs to the DER-S are unlikely to know these mRIDs. Each DER-S will have its own unique identifiers for each input, either provided by the inputs to the DER-S or generated within the respective DER-S class. Each DER-S will also require location data for each of its representative DERs. Using these identifiers and locational data, DER-Ss can be assigned to DER-EMs.

The **DERAssignmentHandler** class automates the task of assigning DER inputs to DER-EMs at the proper location in the model. These locations need to be provided to each DER-S from the DER inputs; they could be the bus the DER should be assigned to, or a member of a more complex topological grouping. Topological processing is handled in **GOTopologyProcessor**, if necessary. Then, during the simulation start-up process, the DERAssignmentHandler determines how many DER-Ss are being used by the system and queries their identification and locational data. Using these data, it assigns each DER-S input's unique identifier to a DER-EM mRID existing at the proper node in the grid model.

The data associating DER-S identifiers, mRIDs, and locational data are stored within the **DERIdentificationManager** class. The inputs to the DER-EMs are handled by the **MCInputInterface**. The MCInputInterface retrieves the electrical state output data from each DER-S and its respective identifier at each timestep, converts the electrical data to a message format usable by the EDM, queries the DERIdentificationManager to replace the DER-S identifier with the respective target mRID for the DER-EM, then sends the messages. These messages are received by the EDM, which updates the BIS models to reflect the new electrical states communicated by the DER-S. These changed grid states are then reflected in the measurements read by the **EDMMeasurementProcessor**.

### 2.3.3 Output Classes

As mentioned above, the **EDMMeasurementProcessor** retrieves grid state measurements once every timestep and appends the locational association data retrieved from DERIdentificationManager. This processed measurement data is sent to two objects at each timestep:

- **MCOutputLog** - Writes the measurements to a log once per timestep for later analysis.
- **GOSensor** - Represents the Grid Operator's sensing system and decision-making process. Filters the measurements as necessary, compares them to user-defined thresholds, and makes determinations as to whether a new grid service should be requested from the DERMS.
  - **GOPostedService** - Grid service requests are packaged into objects of the GOPostedService class. Each of these objects contains attributes holding the service name, group id, type, interval, power, ramp rates, and price. These attributes provide all the information necessary to generate

service request messages for a particular GPostedService object; these objects are maintained in a list in GOSensor and accessed by the GOOutputInterface (see below).

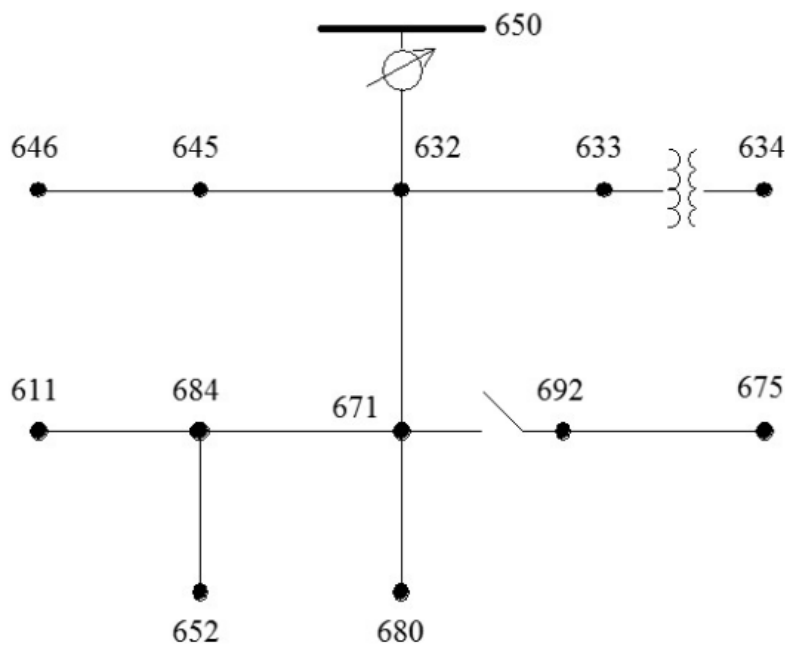
Finally, **GOOutputInterface** provides an interface between the MC and GSP by polling the GOSensor grid service list, packaging grid service requests and feedback data into messages in the proper protocol, and sending them. This interface would be modified for the required needs and protocols if a DERMS other than the GSP were used.

## 3 Implementation

The following sections describe the implementation of the EDM, MC, DER-Ss, and the Grid Operator representation contained within the MC. Information about individual classes including their attributes and methods are included in Appendix B.

### 3.1 Electrical Distribution Model Implementation

The EDM simulates a node-based grid model consisting of generation, loads, and distribution and provides methods to model and control transformers, breakers and switches, reactive power compensation, etc. Rather than develop a complex grid simulator from scratch, we are using GridAPPS-D. The GridAPPS-D system is contained within a Docker container and includes the simulation system, a database of grid models, and a communications bus, which allows inputs and outputs from the system to an external program. As such, GridAPPS-D provides the EDM for the ME system, and the GridAPPS-D python library allows development of an API between the EDM and MC.



**Figure 3.1** The IEEE 13-node test feeder is an example of a node-based model.

#### 3.1.1 Model Database

The GridAPPS-D database contains models in the CIM format. When a simulation is started, GridAPPS-D automatically converts the selected CIM model into a GLM file as part of the startup process.

There are five steps to the process of adding models to the Blazegraph database:

1. The model is generated in a format of the TE's choosing.

2. The model is converted from the above format to CIM via OpenDSS or another comparable software package using CIMHub.
3. The CIM model is ingested to the database.
4. Measurement points are added to the model in the database.
5. The model is validated via CIMHub.

Once a model is contained within the database, DER-EMs (in the form of controllable BIS models) can be added to the model by using CIMHub. CIMHub provides a set of utility functions that allow a model in the database to be modified by the addition of objects, such as houses or DERs, as well as the requisite measurement and control points. These DER-EMs are added to particular nodes in the model and assigned unique mRIDs, allowing them to be associated with DER-Ss in the DER assignment process within the MC.

### 3.1.2 DER-EM Implementation

DERs may function as either load, source, or both. Emulation of thousands of DERs over time is a complex and computationally-intensive task; furthermore, hard-coding the simulator with these emulation processes would unduly couple those specific DER types to the simulator, complicating the process of simulating new types of DERs or modifying the profiles of existing ones with updated models. For this reason, DER-EMs do not contain time-function DER modeling, and instead are generic DERs that respond to control inputs. This allows their modeling to be offloaded to external scripts or to be drawn directly from hardware or sensors. The result of this modeling is a representation of an electrical power source or sink that changes over time; these data are provided to the ME to control DER-EMs.

DER-EMs are implemented in the EDM using BIS models. BIS models exist in the CIM standard and the application can control them using the typical input messaging topics within GridAPPS-D. The MC can control DER-EMs based on inputs from DER-Ss to reflect their electrical effects on the grid over time. Each DER-S receives DER input data from an external file, emulator, or system; these data may represent one or many DERs. Each DER-S is assigned to an appropriate number of DER-EMs automatically during the simulation startup process. This assignment is based on location data taken from the DER-S and matched to location data within the model, allowing DERs to be assigned topologically. Commonly, this location data will be the bus within the model the DER should be assigned to, but more complex topological processing and assignment can be implemented within the GOTOPOLOGYPROCESSOR.

### 3.1.3 EDM-MC Communications

Communications to or from the EDM are handled by the GridAPPS-D message bus. GridAPPS-D has provided a *gridappsd* Python library<sup>3</sup> to simplify communications by packaging requests in intuitive functions. For example, a function exists to write inputs requiring an mRID and state changes as arguments; this function automatically packages the request and sends it to the proper GridAPPS-D API. Another way the library automates communications is by providing “callback functions” or “callback methods” within classes. These functions are called automatically by the simulation at

---

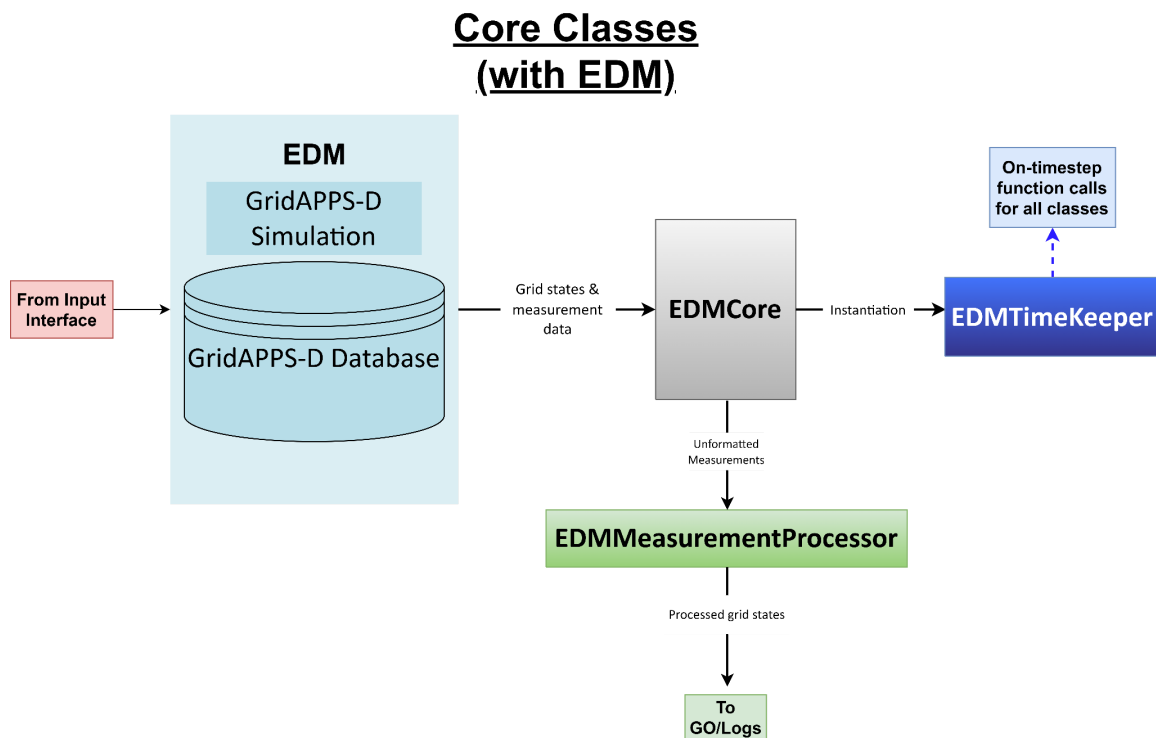
<sup>3</sup> Found at: <https://github.com/GRIDAPPSD/gridappsd-python>

certain times, for instance, when the simulation has updated measurements of grid states, or when a simulation timestep has elapsed. The contents of the callback methods are user-defined. This allows for automation of the MC: processing measurements into a format useful for logs can be performed automatically as the measurements come in, or on-timestep functions can be handled by a callback method that is called once per timestep, for example.

### 3.2 Model Controller Implementation

The MC is the programmed implementation of the ME and is responsible for configuration, execution, and I/O features of the simulation. It is a class-based python script, with each class encapsulating some functions of the system. Organizing the system into a series of class-based actors makes the system more easily extensible. Inputs from new DER emulators and controllers, different logging schemes, varied Grid Operator functions, and external communication schemes can be added, removed, or reconfigured within their respective classes without requiring substantial refactoring outside of the class.

The classes are roughly organized into three groups:

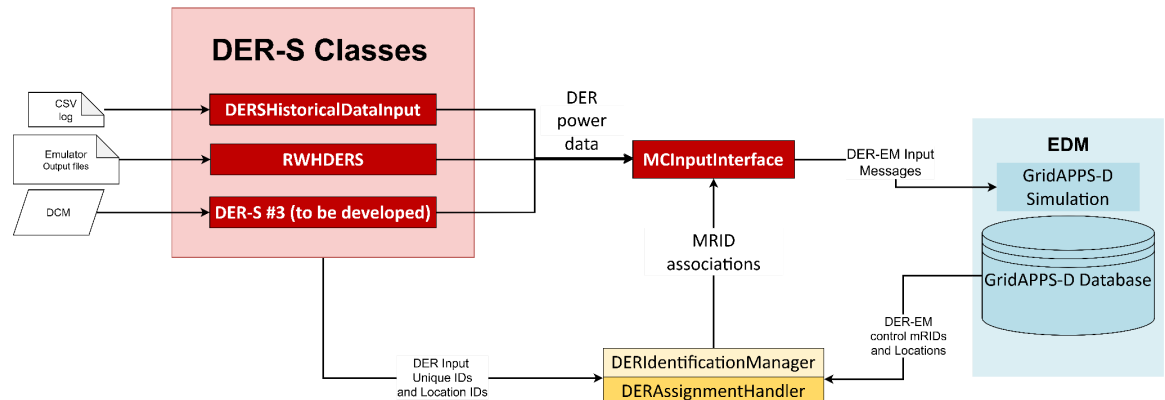


**Figure 3.2** Core classes and their interactions

- **Core classes** govern the simulation configuration and startup, and also include classes with “callback methods,” which are methods that are automatically called by the simulation on startup, once per second, or once per measurement update (roughly three seconds). These callback classes are EDMCore, EDMTimekeeper,

and EDMMeasurementProcessor. MCConfiguration is not a callback class, but is a core class that handles system configuration settings within its attributes.

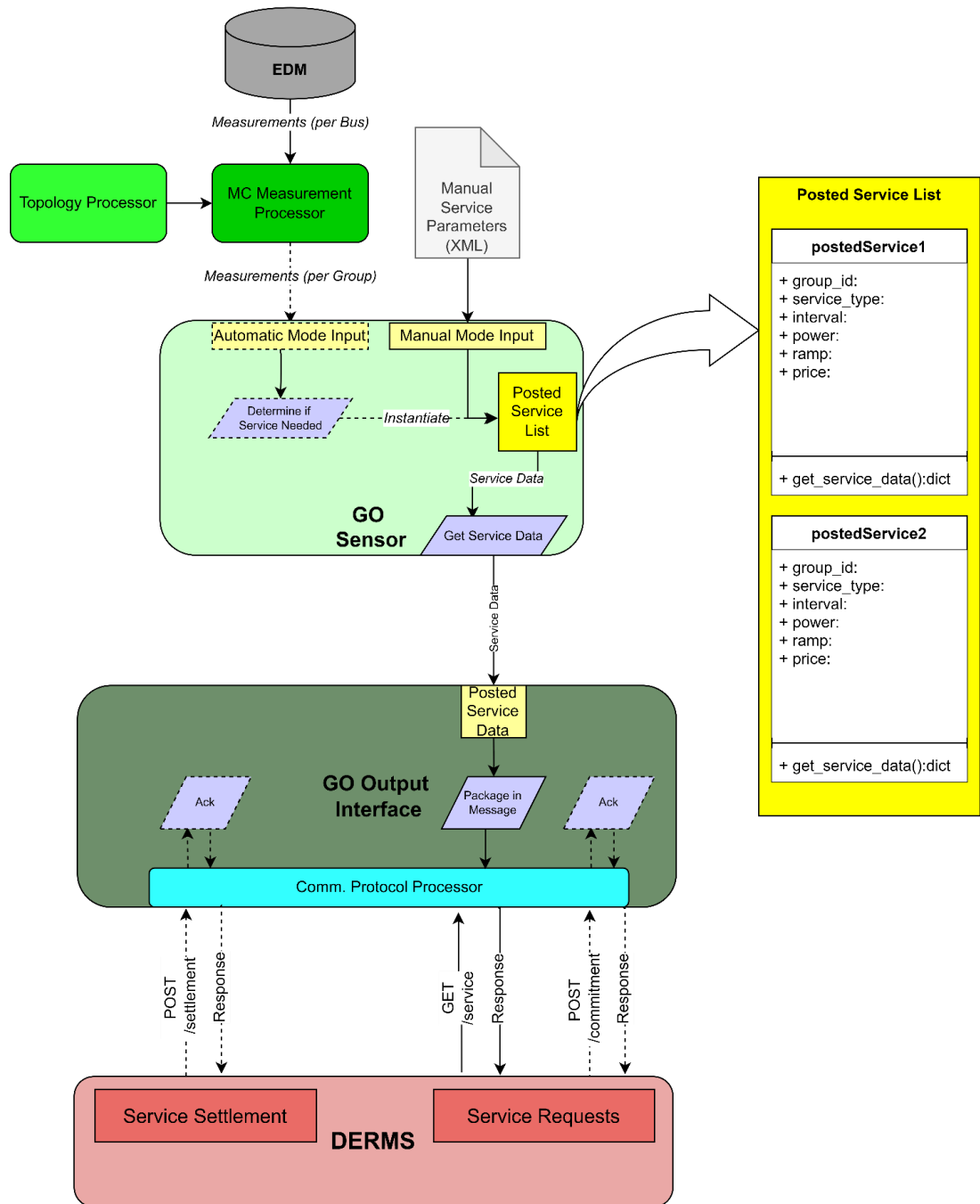
## Input Classes



**Figure 3.3** Input classes and their interactions

- **Input Classes** handle the inputs to the simulation. DER-EMs can be added to the grid model and assigned measurement and control mRIDs prior to the simulation. DER-S classes are added and enabled as required by the needs of the test. The input classes handle communication between the DER-S and MC, assign the unique identifiers of each DER input in each DER-S to the proper DER-EM mRIDs, keep track of the DER-S to DER-EM association data for inputs and logging, and convert DER-S operational changes to input messages for the EDM. These classes are the DER-S interfaces (DERSHistoricalDataInput, RWHDERS, and any other DER-S developed in future), DERAssignmentHandler, DERIdentificationManager, and MCInputInterface.

# Output Classes



**Figure 3.4** The output classes and their interactions, both planned (dashed lines) and implemented (solid lines).

- **Output classes** take the formatted grid state data from EDMMeasurementProcessor and send it to logging or simulated Grid Operator

actors. The logger writes the data to a file for later analysis. The Grid Operator is a simulated actor that functions in one of two modes. The first mode, “automatic mode”, responds to grid state changes by requesting grid services from the DERMS when user-programmed thresholds are met. The second mode, “manual mode,” does not retrieve grid measurement updates but instead generates grid service requests based on an XML input file containing information on what services to request and when.

Grid service requests are objects of the `GOPostedService` class, which contain all necessary information for each grid service request and methods to access this data. These requests (and feedback data) are provided to the DERMS via an interface class: `GOOutputInterface`. The `GOOutputInterface` polls the list of posted service objects once per timestep and accesses the data in any new entries to package them into the proper message format.

Finally, the `GOTopologyProcessor` allows for topological identification. For instance, if the DERMS is configured to register DERs as members of a group of buses rather than a single bus, the topology XML file can be updated to reflect each bus membership in a group, allowing proper assignment, association, and feedback to the DERMS.

The following class diagrams outline each class in terms of attributes and methods. Descriptions of these attributes and methods for each class are included in Appendix B.

Note that this is not a full outline of the system but rather an overview. Many functions and methods have been omitted, such as file system, data processing, and most accessor methods.

### **3.2.1 Model Controller Core Classes**

The core classes include `EDMCore`, `EDMTimeKeeper`, `EDMMeasurementProcessor`, and `MCConfiguration`. Class diagrams for each are shown in Figure 3.5.



**Figure 3.5** UML class diagrams of the callback classes.

The **EDMCore** class provides EDM configuration and simulation startup functionality. It also instantiates the other callback classes that are used to handle the EDM to MC communication pipeline.

The **MCConfiguration** class contains configuration settings for the Model Controller in its attributes, making it convenient for the user to configure paths, file names, and active DER-S classes.

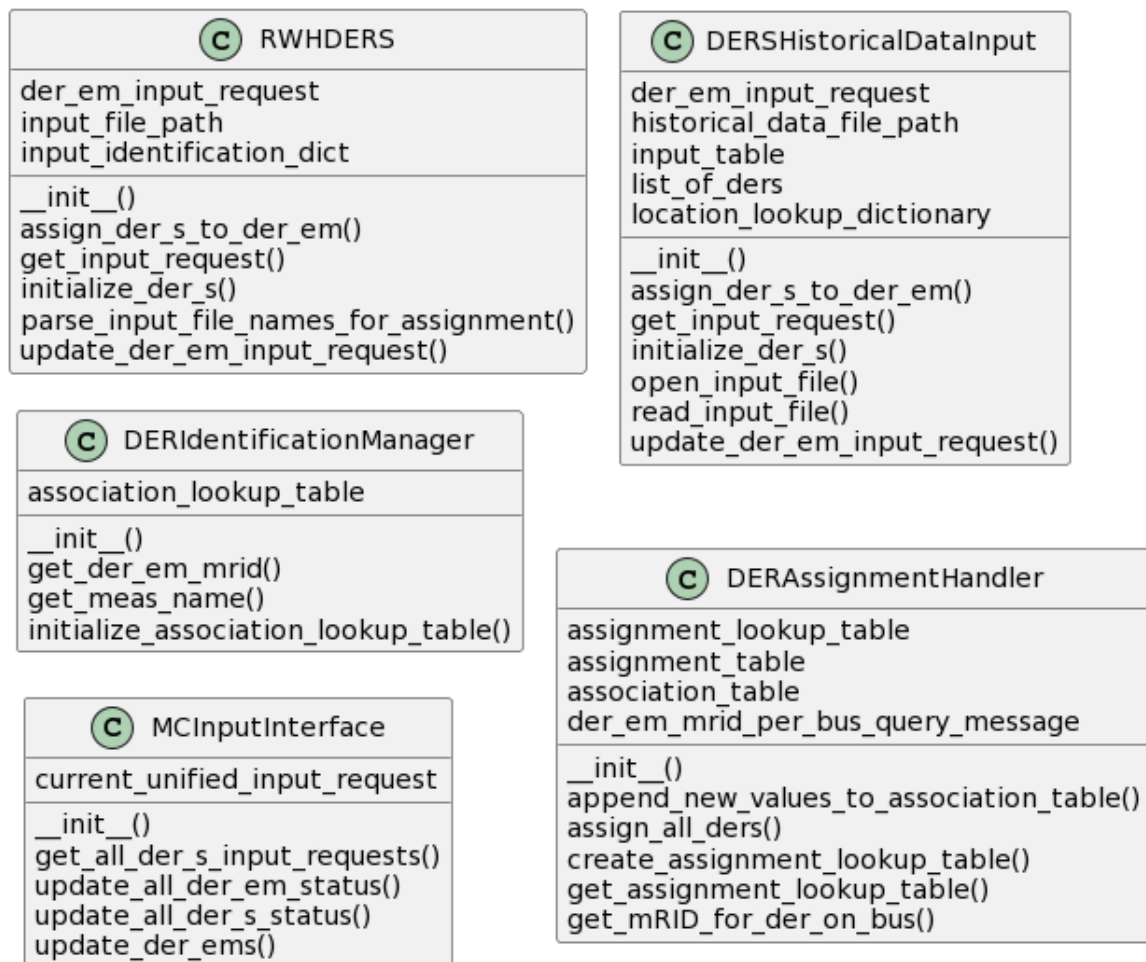
The **EDMTimeKeeper** callback class receives a message from GridAPPS-D once per simulation timestep, which is one second. On receipt, the class updates the global simulation time and sends that to any actors that need it. This class also instructs actors

to update themselves if they are meant to do so once per timestep (such as the MCInputInterface), rather than once per measurement (such as the MCOutputLog)

The **EDMMeasurementProcessor** callback class operates once every three seconds. GridAPPS-D sends a message to this class containing a dictionary with a timestamp along with dictionaries containing measured quantities from the simulation. These measurements are keyed by mRID; they do not include human-readable information such as the names, locational data, etc. This class draws said information from the lookup tables in DERIdentificationManager, reformats the message into a single “row” of data for the logs, and sends it to MCOutputLogs and the GOSensor classes.

### 3.2.2 Model Controller Input Classes

The Input classes include the DERIdentificationManager, DERAssignmentHandler, and MCInputInterface. Also included are a number of customizable DER-Ss to handle inputs to the system. Two examples are provided: the Resistive Water Heater DER-S class (RWHDEERS), and a general-purpose historical data log processor class, DERSHistoricalDataInput. Class diagrams for each are shown in Figure 3.6.



**Figure 3.6** UML class diagrams of the input classes.

The Resistive Water Heater DER-S class **RWHDERS** is one example of a DER-S interface within the ME. This class reads emulated water heater values from an input buffer, which in the current configuration is a folder containing CSV files representing each DER that are updated on a running basis by the GSP. Once per timestep, RWHDERS reads the files, parses their contents for their unique identifier and electrical states data, and provides that data to the MCInputInterface.

The **DERSHistoricalDataInput** class represents a DER-S that is defined by timestamped historical data from a CSV file. The input CSV must provide unique identification data for each DER, as well as location information (such as a bus) for each DER. This allows the DER inputs to be assigned to DER-EMs like any other DER-S.

Time	DER1_Watts	DER1_VARS	DER1_loc	DER2_Watts	DER2_VARS	DER2_loc	DER3_Watts	DER3_VARS	DER3_loc
1570041118	0	0	632	0	0	633	1000	634	645
1570041119	0	0	632	1000	0	633	0	634	645
1570041120	0	0	632	1000	0	633	0	634	645
1570041121	0	0	632	1000	0	633	0	634	645
1570041122	0	0	632	1000	0	633	1000	634	645
1570041123	0	0	632	1000	0	633	1000	634	645
1570041124	0	0	632	1000	0	633	1000	634	645

**Table 3.1** Example data from a DERSHistoricalDataInput CSV file. Timestamps are in unix time format. Each trio of columns serves a single DER Input; DER1\_Watts, DER1\_VARS, and DER1\_loc are the real power (in Watts), the reactive power (in VARs), and the locational identifier (or bus, in this case). The DER Input's unique identifier is taken from the header, and in this case is "DER1\_mag". The unique and locational identifiers are used for assignment.

**DERIdentificationManager** class handles lookups for all association data in the simulation. 'Association data' refers to the tables that correlate each DER input with its associated DER-EM; this includes each representative unique identifier from the DER inputs (as handled by the DER-S), mRIDs of DER-EMs in the simulation used by the MCInputInterface, and locational data. Identification data are used primarily by the MCInputHandler to send input messages to the correct DER-EM using only identification data, or by the log for identification purposes during testing.

The **DERAssignmentHandler** class manages the creation of the DER association lookup table. It differs from the DERIdentificationManager in that it is only meant to be used in the ME startup process. The role of the assignment handler is to take mRIDs and location data from the GridAPPS-D database, associate them with provided unique

identifiers for each DER-S, and provide that data to the DERIdentificationManager in the form of a table.

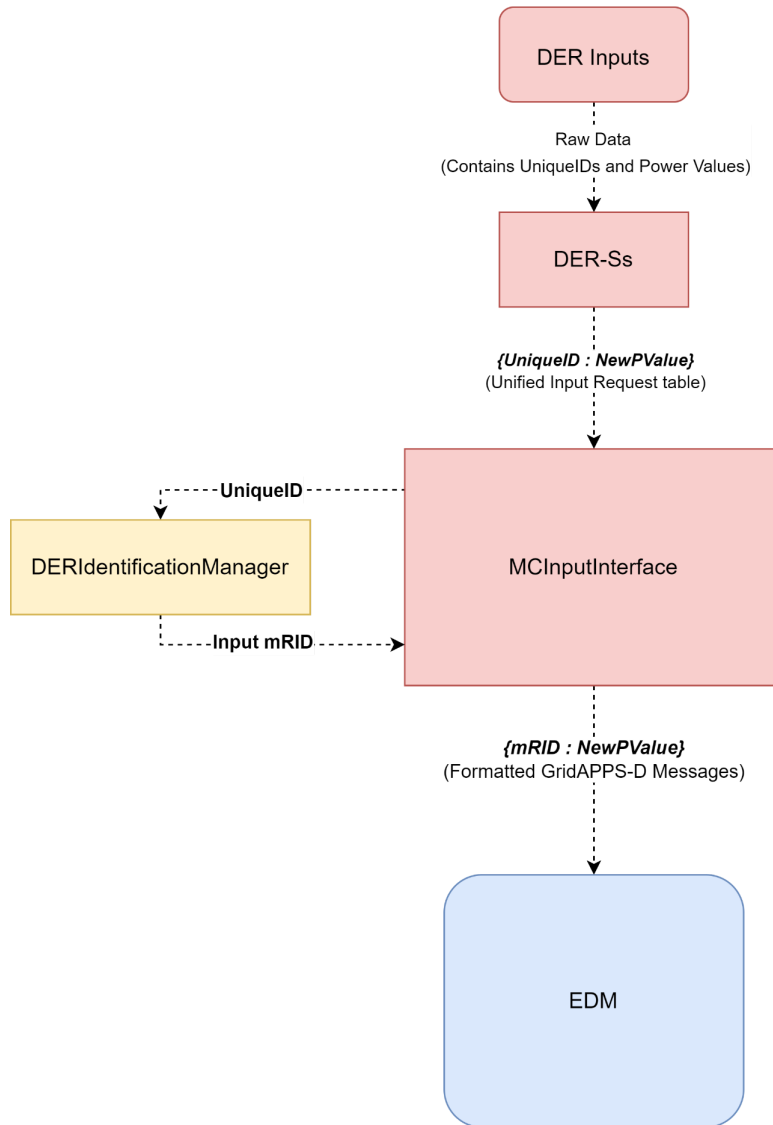
### DER Assignment Process



**Figure 3.7** DER Input unique and locational identifiers are used to assign a DER input to a DER-EM by its control mRID.

The **MCInputInterface** class injects updated DER states into the model. Each timestep, it retrieves updated data from each DER-S, packages it into formatted messages, and delivers the messages to the EDM via the GridAPPS-D input topic.

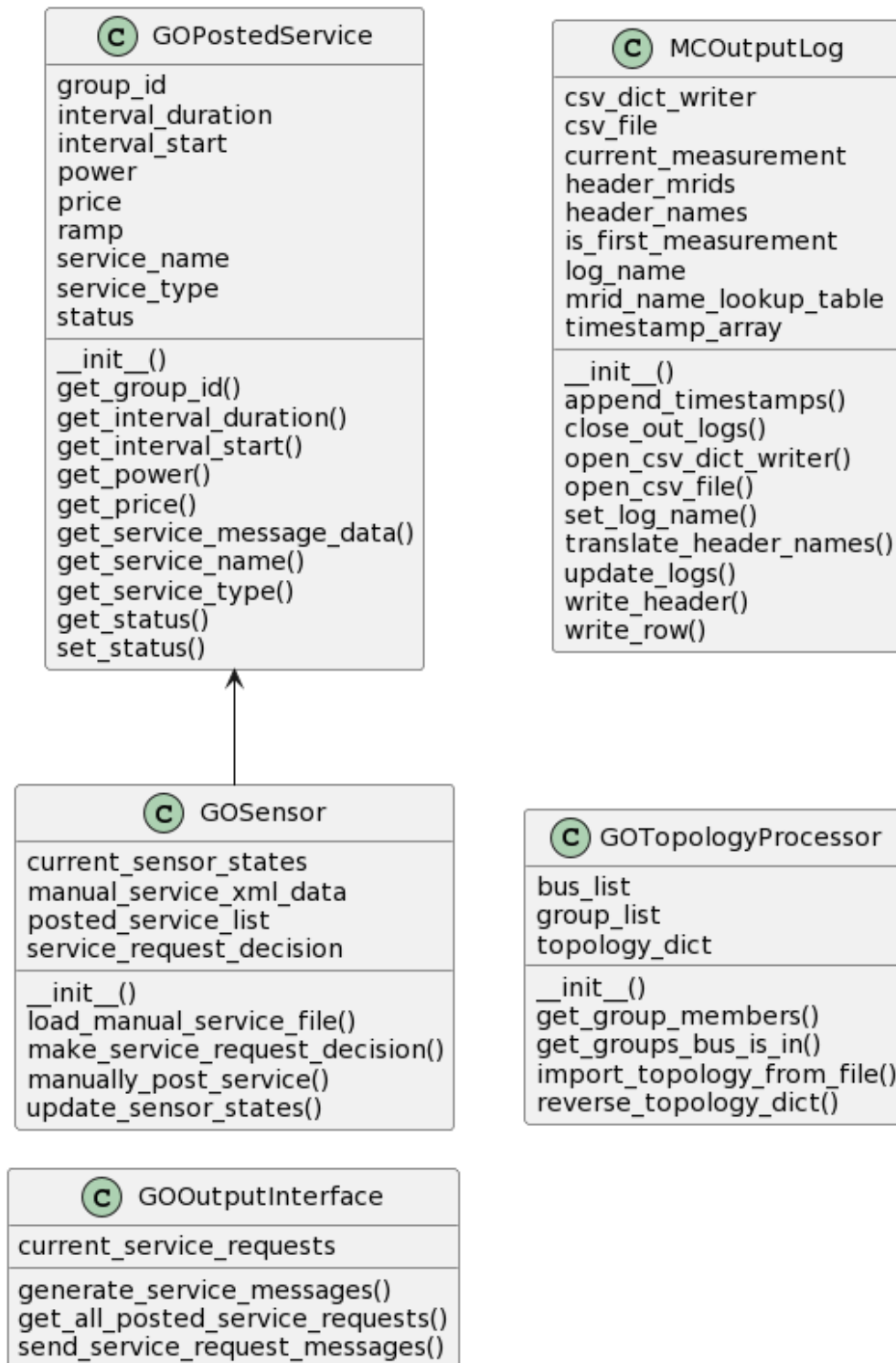
### Input Message Process



**Figure 3.8** After assignment, the association data is held within the DERIdentificationManager, automating the process of looking up DER-EM control mRIDs.

### **3.2.3 Model Controller Output Classes**

The Output classes include GOSensor, GOOutputInterface, MCOutputLog, and GOTopologyProcessor. Another output class is GOPostedService, which is unique in that it is the only class that may be instantiated into more than one object. Class diagrams for each are shown in Figure 3.9.



**Figure 3.9** UML class diagrams of the output classes.

The **GOSensor** class represents the operations of the Grid Operator and is meant to be configurable for the needs of the model or the test. There are two possible operating

modes: automatic mode and manual mode. In automatic mode, GOSensor reads in grid states from the measurement processor (hence the name “GOSensor”); these are compared to thresholds based on a user-defined algorithm and service requests are automatically generated. For example, the GOSensor detects voltage deviations beyond 5% and, as a result, generates a Voltage Management service request to DERMS. In manual mode, a list of services, parameters and request times are read from a file during system startup. The GOSensor then generates service requests with the proper parameters at the designated times. This mode is sufficient for most functional testing purposes and is considered the “default” mode of operation. In either case, service requests are generated by the instantiation of an object of the GOPostedService class; these objects contain the service request parameters, are kept in a list by GOSensor and accessed by the GOOutputInterface.

The **GOOutputInterface** class handles service request messaging between the MC and the DERMS. It makes connections as necessary, polls the GOSensor for new service requests each timestep, accesses posted service request parameters, packages them into the proper message format, and sends the messages to the DERMS. The actual decision-making process is in GOSensor.

The **MCOutputLog** class handles logging. Processed measurements from EDMMeasurementProcessor are retrieved, converted to dictionary format, and written to a CSV file for later data analysis.

Timestamp	PowerTransformer_sub3_Power	PowerTransformer_sub3_Power.1
2019-10-02 18:31:58	{'measurement_mrid': '_01e96721-222d-42be-bcc3-8cb6fe23d44c', 'magnitude': 1474276.2673145642, 'angle': -11.711333830924843, 'Measurement name': 'PowerTransformer_sub3_Power', 'Meas Name': 'PowerTransformer_sub3_Power', 'Conducting Equipment Name': 'sub3', 'Bus': '650', 'Phases': 'B', 'MeasType': 'VA'}	{'measurement_mrid': '_122affee-e7dd-4d8f-bd10-2696fd95c950', 'magnitude': 1415691.1267786373, 'angle': -7.507786248792342, 'Measurement name': 'PowerTransformer_sub3_Power', 'Meas Name': 'PowerTransformer_sub3_Power', 'Conducting Equipment Name': 'sub3', 'Bus': '650z', 'Phases': 'A', 'MeasType': 'VA'}
2019-10-02 18:31:59	{'measurement_mrid': '_01e96721-222d-42be-bcc3-8cb6fe23d44c', 'magnitude': 1474276.2673145642, 'angle': -11.711333830924843, 'Measurement name': 'PowerTransformer_sub3_Power', 'Meas Name': 'PowerTransformer_sub3_Power', 'Conducting Equipment Name': 'sub3', 'Bus': '650', 'Phases': 'B', 'MeasType': 'VA'}	{'measurement_mrid': '_122affee-e7dd-4d8f-bd10-2696fd95c950', 'magnitude': 1415691.1267786373, 'angle': -7.507786248792342, 'Measurement name': 'PowerTransformer_sub3_Power', 'Meas Name': 'PowerTransformer_sub3_Power', 'Conducting Equipment Name': 'sub3', 'Bus': '650z', 'Phases': 'A', 'MeasType': 'VA'}
2019-10-02 18:32:00	{'measurement_mrid': '_01e96721-222d-42be-bcc3-8cb6fe23d44c', 'magnitude': 1474276.2673145642, 'angle': -11.711333830924843, 'Measurement name': 'PowerTransformer_sub3_Power', 'Meas Name': 'PowerTransformer_sub3_Power', 'Conducting Equipment Name': 'sub3', 'Bus': '650', 'Phases': 'B', 'MeasType': 'VA'}	{'measurement_mrid': '_122affee-e7dd-4d8f-bd10-2696fd95c950', 'magnitude': 1415691.1267786373, 'angle': -7.507786248792342, 'Measurement name': 'PowerTransformer_sub3_Power', 'Meas Name': 'PowerTransformer_sub3_Power', 'Conducting Equipment Name': 'sub3', 'Bus': '650z', 'Phases': 'A', 'MeasType': 'VA'}

**Table 3.2** Subset of data contained within an output log file. Note that each cell contains a dictionary of data which can be used or filtered out during analysis.

**GOTopologyProcessor** allows for more complex topological assignments. A topology file is read into the object on startup. This file must be user-defined to match the topology expected by the DERMS and contains XML dictionaries clustering service points (buses) into groups. This information can then be used during the DER assignment process, allowing groups to be entered as locational identifiers. These groups are translated to appropriate buses, which are fed back to the assignment handler to continue the assignment process.

Our team customized the IEEE 13-Node feeder shown in Figure 3.1 to simulate  $\approx 1000$  DERs. The topology of this feeder is designed to align with the Common Smart Inverter Profile (CSIP) topological grouping scheme<sup>4</sup>. The PSU IEEE-13 Node topology incorporates three groups; each group includes feeders, segments, transformers, and service points, as shown in Figure 3.10. Such configuration provides flexibility within the **GOTopologyProcessor**, allowing for a wide range of grid services levels. The entire topology of the PSU IEEE 13-Node feeder is available on GitHub<sup>5</sup>.

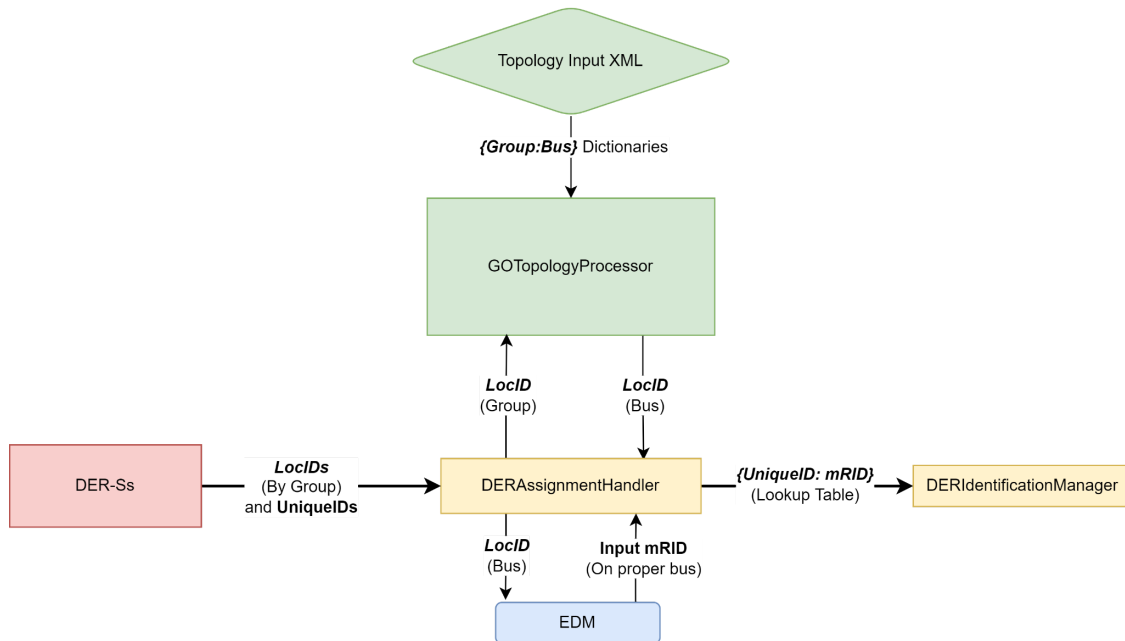
```
<Groups>
  <group name= "group-1" >
    <Feeder name= "OL630-632" >
      <Segment name= "UL632-633" >
        <Transformer name= "xfmr_633_a_1" >
          <ServicePoint name= "tlx_633_a_h_1"/>
          <ServicePoint name= "tlx_633_a_h_2"/>
          <ServicePoint name= "tlx_633_a_h_3"/>
          :
        </Transformer>
        <Transformer name= "xfmr_633_a_2">
          <ServicePoint name= "tlx_633_a_h_9"/>
          <ServicePoint name= "tlx_633_a_h_10"/>
          <ServicePoint name= "tlx_633_a_h_11"/>
          :
        </Transformer>
        <Transformer name= "xfmr_633_a_3">
          <ServicePoint name= "tlx_633_a_h_17"/>
          <ServicePoint name= "tlx_633_a_h_18"/>
          <ServicePoint name= "tlx_633_a_h_19"/>
          :
        </Transformer>
      </Segment>
    </Feeder>
  </group>
</Groups>
```

**Figure 3.10** Customized IEEE 13-Node Feeder topology aligning with the CSIP grouping scheme. This is an example of only “group” out of three groups.

<sup>4</sup> Common Smart Inverter Profile Working Group. [Common smart inverter profile v2.1](#).

<sup>5</sup> [Portland State University Version of IEEE 13-Node Feeder](#)

### Example Topological Processing (During Assignment Process)



**Figure 3.11** During assignment, the GOTopologyProcessor translates non-bus locational identifiers to buses using a predetermined topology (see Figure 3.10)

**GOPostedService** is the only class in the MC that is instantiated more than once; the GOSensor class creates an object each time it determines that a service should be requested. All service request parameters are included in attributes of the GOPostedService object; this includes a name, a group id, the service type, interval, and parameters. These objects are contained within a list in the GOSensor class and queried by the GOOutputInterface once per timestep.

### 3.3 DER-S Implementation

The ME is designed with extensibility in mind, particularly regarding the ability to implement models of a variety of DERs, both existing and those yet to be developed. Each DER-S is a black box: it receives data from one or more DER inputs, and provides unique and locational identifiers (for assignment and association) and power sourced or sunk (for DER-EM control). However, the contents of the DER-S are deliberately undefined. This provides a greater amount of flexibility in how a particular DER-S can be developed and allows for hardware-in-the-loop implementations, inputs from emulators, and historical data to be used. Of course, this also means that any type of DER can be included in the MC as long as a DER-S is developed for said DER, along with any sensors, instruments, emulators or programs required to model said DER externally to the ME.

Due to this flexibility, it is impossible to provide an exhaustive list of possible implementations, particularly with regards to inputs or internal processing within a

DER-S. Some examples are presented in Sections 3.3.1 and 3.3.2. Note that each example refers to one or more DER-S classes that would need to be developed for a particular purpose or DER.

### **3.3.1 DERMS to DER-S communication**

Typically, information exchange between the DERMS and DER is beyond the scope of the ME since the DER-S is concerned with reading the operating state of a DER, not how that state is achieved. However, it is conceivable that for test purposes a DERMS could request resources that would be convenient to implement directly within the MC. One example would be a DERMS that controls BISs directly.

In this example, communications between the DERMS and DER-S would need to be established via development of an API within the DER-S class that handles and parses these communications. This is typical for all DER-S types; in this particular case, however, the communication protocol would be dictated by the DERMS. After the communications are parsed, they will be processed into the proper output format within the DER-S class.

#### **3.3.1.1 DERMS to DER-S using Physical Systems**

A physical DER will operate per normal operating procedures most of the time. At times, however, it will request grid-service participation with the DERMS. This messaging requires a communication pathway between the DERMS and DER via the DER control system, such as a Distributed Control Module (DCM). The DERMS-to-DCM information exchange is managed by the DERMS, and is beyond the scope of the ME.

In this case, the DER-S class will interface with one or more DCMs by a communications API between the DCM and DER-S class. The contents of these communications would be dictated by the capabilities of the DCM, and the DER-S would govern processing these communications into a readable by the MCInputProcessor and DERAssignmentHandler classes. For example, one method of modeling a DER is to control its power consumption over time. A sophisticated enough DCM with the proper sensors in its DER could provide power consumption readings directly to the DER-S, along with unique identifiers. A more rudimentary DCM may only provide a flag that the DER is “turned on” to the DER-S class, which would require the DER-S to “emulate” power consumption patterns based on the DER nameplate information. Locational identifiers would likely need to be provided to the DER-S by the test engineer, using an input configuration file or an algorithm for automatic generation.

Due to the flexibility in DER-S design, no communication protocols or standards are suggested. These should be selected based on the needs of the particular DERMS, DCM, and DER-S implementation in use.

#### **3.3.1.2 DERMS to DER-S using Emulated Systems**

DER-S implementations that do not include physical hardware should instead use software algorithms to generate operating data. This need not be done within the DER-S class: sophisticated external emulators or functions within the DERMS may be used for this purpose, handling all communications and time-valued DER modeling. The DER-S

class needs to extract some information from these emulators that can be parsed into electrical data. These “DER inputs” may be electrical information, such as power consumption, that may be fed directly into the simulation. Or, they may be more generic operating data that are converted to electrical information within the DER-S by the inclusion and processing of label plate data or similar parameters.

One example of an emulated DER<sup>6</sup> that has been developed is a resistive water heater, developed specifically to test the GSP. One or many resistive water heaters are emulated within the GSP; the power consumption of each water heater is placed in a constantly updated file containing unique identifiers, location data, and operation data in the form of power. These files are read in real time by the RWHDEERS, a DER-S class written for this purpose. While RWHDEERS provides a power reading directly, another DER-S implementation may not. One conceivable DER-S would function similarly to RWHDEERS but instead of providing power readings, it may only provide whether the water heater is “importing” or “exporting” power. The import/export flags would be translated into power consumption data by the DER-S using provided label plate ratings. The role of the DER-S class is to read the data, perform the operational/electrical power conversion, and pass those data in the proper form to the MCInputInterface for use in DER-EM updates.

### 3.3.2 DER Modeling using Historical Data

“Historical data” in this case refers to any input to the simulation that is not being generated in real-time. For example, this could be data generated by an emulator, or it could be states read from DCM sensors and written to a file. In any case, a DER-S needs to be written or configured to parse the historical data and provide it to the simulation at appropriate times. These data must be time stamped to be used at the proper time in the simulation, and must include unique and locational identifiers so DERs in the data can be assigned to DER-EMs in the EDM. This is a preferred method for generating inputs to the DER-EMs in many functional test cases due to its simplicity: direct communications do not need to be established between the DERMS/emulators and DER-S during simulation runtime and large amounts of data can be processed without encountering communication obstacles such as bandwidth and latency.

Time	DER1_mag	DER1_loc	DER2_mag	DER2_loc	PowerElectronicsConnection_Battery Unit_DER_Association_Test_6321_Battery.3	PowerElectronicsConnection_Battery Unit_DER_Association_Test_6331_Battery.3
1570041118	0	632	1000	633	{ 'magnitude': 1333.333333, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1666.666667, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041119	1000	632	0	633	{ 'magnitude': 1333.333333, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1666.666667, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041120	1000	632	1000	633	{ 'magnitude': 1333.333333, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1666.666667, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }

<sup>6</sup> <https://github.com/PortlandStatePowerLab/doe-egot-dcm/tree/main/src/ecs/include/ecs>

1570041121	0	632	0	633	{ 'magnitude': 1333.333333, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1333.333333, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041122	0	632	1000	633	{ 'magnitude': 1333.333333, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1333.333333, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041123	1000	632	0	633	{ 'magnitude': 1333.333333, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1333.333333, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041124	1000	632	1000	633	{ 'magnitude': 1666.666667, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1666.666667, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041125	0	632	0	633	{ 'magnitude': 1666.666667, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1666.666667, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }
1570041126	0	632	1000	633	{ 'magnitude': 1666.666667, 'Bus': '632', 'MeasType': 'VA', 'Input Unique ID': 'DER1_mag' }	{ 'magnitude': 1666.666667, 'Bus': '633', 'MeasType': 'VA', 'Input Unique ID': 'DER2_mag' }

**Table 3.3** An example of the effects of DER inputs on grid state data over time. The inputs are drawn from an input CSV file for the DERHistoricalDataInput DER-S, and the outputs are log outputs that have been edited for clarity. Note that each column represents a single phase; as such, an added 1 kW of load shows as 333 W per phase. Also note the discrepancy between simulation and measurement timesteps: while inputs are delivered once per second, the measurements only update once every three.

### 3.4 Grid Operator Implementation

As a component of the ME, the GO is a set of classes that simulate the decision-making process and actions of a real Grid Operator. The GO also provides an interface for communicating grid service requests and feedback data to the DERMS, and provides a method for topological translation. The GO will have two modes of operation: Automatic and Manual mode. Automatic mode is designed to realistically simulate GO operations by forecasting the demand profiles and retrieving grid states once per timestep, making determinations based on the grid states and built-in detection algorithms, and posting service requests automatically based on grid conditions. In Manual mode, grid service requests are read in from an XML file and posted at the proper times, allowing for simple, direct control for functional testing.

In a fully functional test system such as the EGoT prototype, during a running simulation the GOSensor may request a service from the DERMS via the GOOutputInterface. Externally to the ME, the DERMS will dispatch DERs per this request. These changes in DERs will be reflected by the DER-EMs within the EDM via the respective DER-S interface. These changes in the DER-EMs affect the grid states, which are gathered by the EDMMeasurementProcessor and (in Automatic mode) sent to the GOSensor. The GOSensor will parse the EDM grid states to verify the dispatch had occurred and incurred the required grid effects. The GO is meant to be a high-level representation of the GO's decision making; as such, the GOSensor decision-making function is meant to be a simple threshold detection script, though the system can be updated with more advanced detection capabilities if necessary. The GOOutputInterface communication

with the DERMS is meant to be configured by the end user rather than requiring a specific protocol.

### **3.4.1 Grid States Inputs**

During the simulation process, the EDMMeasurementProcessor holds a table containing the most recent grid states read from the ME at each time step, including electrical characteristics, the status of each DER-EM, and its association data (mRIDs, locational data, etc.) along with any other relevant grid state data. These tables are sent to the MCOutputLog once per time step to construct the logs. If Automatic Mode is enabled, they are also sent to the GOSensor, which parses them. The data may be limited by the GOSensor to better simulate a real-world GO, or it may simply read in all grid states data directly.

### **3.4.2 Decision-Making Process**

There are two types of tests involving grid services that the TE may wish to conduct. First, the TE may want to test the effects of a grid service on a typical grid, to gather data on how a service affects a grid. Second, the TE may wish to test the GSP's ability to correct actual problems by dispatching DERs. In the former test, the GO can be set to manually request a grid service per the TE's needs; in the latter, the GO may need to make that determination automatically.

#### **3.4.2.1 Manual Decision-Making Mode**

In this mode, the TE will select a particular desired request and a specific time to send the request. This will be done for one or more requests by placing the service request parameters and times in a properly formatted XML file and configuring the MC to read it in during the startup process. As the simulation is running, the GO will keep time with the simulation using the MCTimekeeper current\_time attribute. At the programmed time, the GOSensor will generate a GOPostedMessage object containing the parameters of the grid service and place this object in a list. Each timestep, the GOOutputInterface reads this list and, if a new object is found, generates a message in the proper format containing the service parameters. The GO will then send the message to the DERMS. The format and protocol of this message will be dependent on the needs of the DERMS; generally, the message will contain a request for a particular service and any related information such as durations, levels, etc. This mode is suitable for testing the output of the DERMS and its effects on the grid without necessarily requiring an abnormal grid condition to exist.

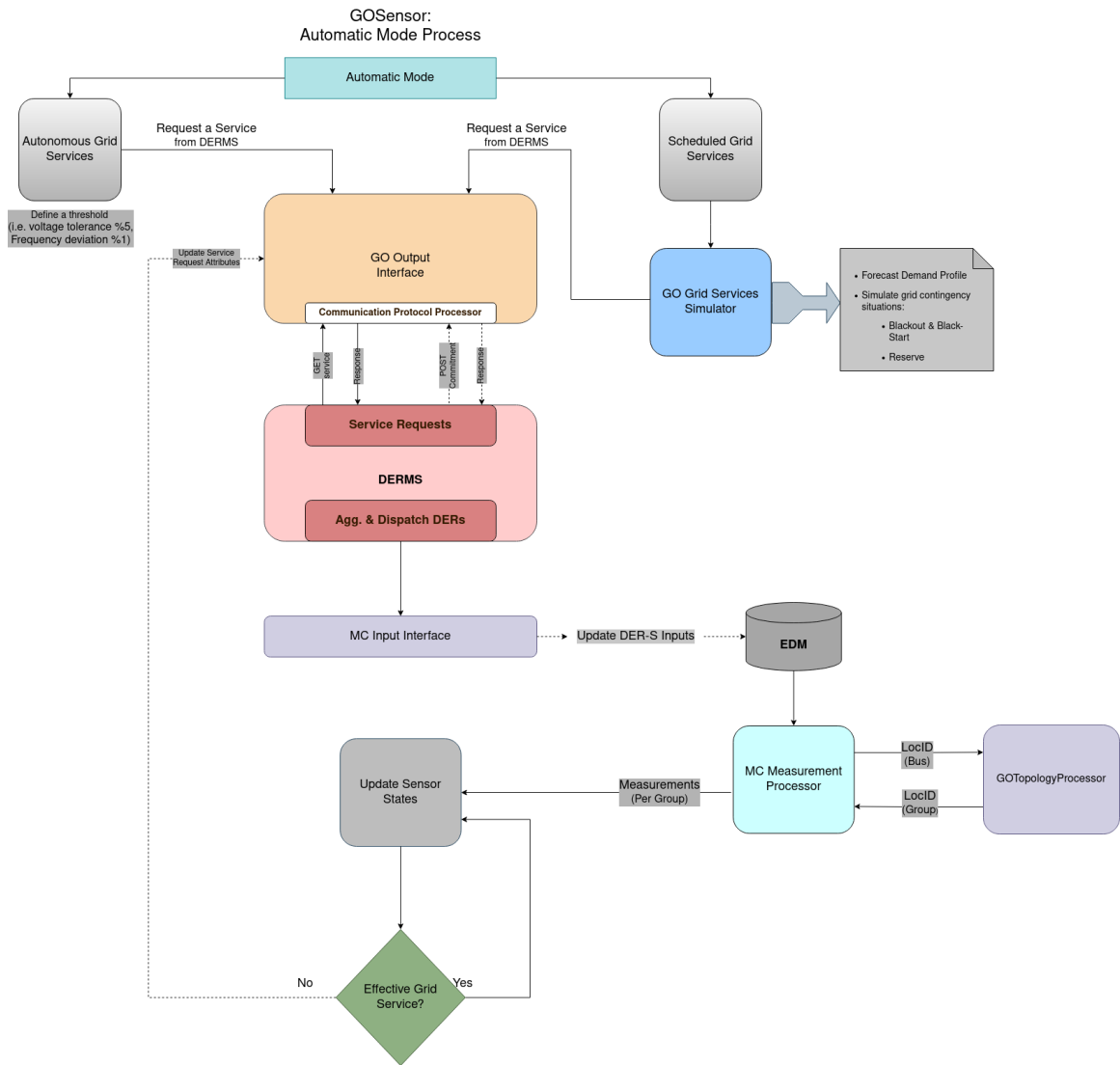
#### **3.4.2.3 Automatic Decision-Making Mode**

In this mode, the TE does not request a service at a particular time. Instead, the GO is configured to forecast the demand profiles and schedule a service based on the forecasted data. The GO is also configured to measure a particular component or value via the feedback function and compares it to a programmed threshold once per time step. If the threshold is exceeded, a GOPostedService object is generated, and the process is followed identically to a manual mode.

Figure 3.11 depicts the GO automatic Decision-Making mode operation. In this mode, grid services are divided into two categories: autonomous and scheduled grid services. The former method is meant for grid services requiring DER capabilities that respond automatically to grid variations, such as Voltage management and Frequency Response. The autonomous grid services method defines frequency and voltage threshold values for measurements read by the GOSensor. Simultaneously, the autonomous grid services method requests a service from DERMS to aggregate DERs with suitable algorithms to provide the requested service. In each timestep, the GOSensor evaluates the measurements against the defined thresholds; if the thresholds are exceeded, the GO updates the service attributes (location and time interval), indicating that the dispatched DERs did not meet the requested service requirements. Otherwise, the GOSensor monitors the grid states for the service interval to ensure the measures taken by DERMS meet the requested grid service objectives. For instance, in a voltage management service, the DERMS updates aggregated DERs with Volt-VAr curve points prior to the service time. As such, the DERs respond to voltage excursions beyond the defined threshold by injecting or absorbing reactive power.

The scheduled grid services method simulates the GO behavior with grid services that are planned in advance, such as blackstart, reserve, and energy grid services. At the beginning of the simulation, the GO forecasts the demand profiles and simulates a grid contingency situation (i.e., blackout or sudden change in load). When forecasting the demand profiles, the GO reads data provided by the **DERSHistoricalDataInput** class and identifies the peak demand periods. The GO then requests a service from DERMS, either a peak demand shifting or mitigation, based on the outputs of the forecasting process. As the service time starts, the GO monitors the grid states and ensures that DERMS's actions realize the requested service's objectives.

When simulating a grid emergency event, the **Grid Services Simulator** class packages the event's start time, duration, end time, and topological location and send it to the DERMS through the **GOOutputInterface** at the beginning of the simulation. The role of the **Grid Services Simulator** is to simulate the event at the scheduled time. For instance, to test the DERMS effectiveness in responding to blackstart grid service, the **Grid Services Simulator** class simulates a blackout during the scheduled service time by opening a switch that disconnects a feeder from the source. The GO interacts with the DERMS during the blackout to update the feeder restoration time to revert DERs to normal operation.



**Figure 3.11:** GOSensor Automatic Mode Operational Scheme.

### 3.4.3 GO-DERMS Communications


Since the ME is designed to be used with a variety of DERMSs, there is no single protocol for information exchange between a GO and a DERMS. This will necessarily be defined by the user within the GOOutputInterface class. To simplify this process, all results of the decision-making process are simplified into “grid service requests” and “feedback data.” As such, the TE will need to develop an API between the GO and the DERMS in order to translate request types and magnitudes into messages that both actors can understand.

## Appendix A

### A.1 MC Callback Classes

#### A.1.1 EDMCore Class

“““Provides central, core functionality to the MC. Responsible for the startup process and storing the GridAPPS-D connection and simulation mRIDs and objects.”””

 EDMCore
<div><div>cim_measurement_dict</div><div>config_parameters</div><div>gapps_session</div><div>line_mrid</div><div>mrid_name_lookup_table</div><div>sim_current_time</div><div>sim_mrid</div><div>sim_session</div><div>sim_start_time</div></div>
<div><div><div>__init__()</div><div>connect_to_gridapps()</div><div>connect_to_simulation()</div><div>create_objects()</div><div>establish_mrid_name_lookup_table()</div><div>get_cim_measurement_dict()</div><div>get_line_mrid()</div><div>get_mrid_name_lookup_table()</div><div>get_sim_start_time()</div><div>initialize_all_der_s()</div><div>initialize_line_mrid()</div><div>initialize_sim_mrid()</div><div>initialize_sim_start_time()</div><div>load_config_from_file()</div><div>sim_start_up_process()</div><div>start_simulation()</div></div></div>

##### A.1.1.1 EDMCore Attributes

**gapps\_session** The gridappsd library allows the script to connect to the GridAPPS-D simulation running in the docker container. That function returns an object that is stored in this attribute for use in direct queries and later when initializing the simulation.

**sim\_session** This object is returned by the gridappsd library simulation() method when the gapps\_session object and the configuration parameters are passed in. This object represents the current simulation (rather than the environment as a whole in

gapps\_session) and is used to instantiate the subclasses and get the simulation ID, providing general simulation control.

**sim\_start\_time** The unix timestamp of the simulation start time. This is the start time that the simulation will increment from until reaching the end of the simulation duration and is not associated with the real time.

**sim\_current\_time** This is running the most current timestep of the simulation. This is the variable used for global timekeeping purposes.

**sim\_mRID** The Simulation ID of the current simulation.

**line\_mRID** The mRID of the model being simulated. This can be used to query various aspects of the model (such as measurement points) directly from the blazegraph database. GeoRegion and SubGeoRegion are contained within config\_parameters as well, but aren't given attributes since they're never needed for our purposes.

**config\_parameters** A dictionary of configuration parameters required by GridAPPS-D to start a simulation. Includes the line model mRIDs, application configurations as needed, and simulation configuration including the start time, duration, timestep, and other options required by GridAPPS-D during startup.

**mrid\_name\_lookup\_table** Object measurement table queried from the database. Contains information connecting mRIDs to names for measurements. These are used by the logs to make headers more human-readable.

**sim\_measurement\_dict** A similar, but separate table queried in a different way from mrid\_name\_lookup\_table. This measurement dict contains amplifying data for measurements, such as the measurement type or bus location. These are added to the logs to make data cells more human-readable.

#### A.1.1.2 EDMCore Methods

**get\_sim\_start\_time()** ACCESSOR METHOD: returns the simulation start time (per the configuration file, not realtime)

**get\_line\_mRID()** ACCESSOR METHOD: Returns the mRID for the current model.

**sim\_start\_up\_process()** ENCAPSULATION METHOD: calls all methods required to set up the simulation process. Does not start the simulation itself, but performs the “startup checklist.” This includes connecting to GridAPPS-D and the simulation, loading configuration from the file, instantiating all the (non-callback) objects, initializing DER-Ss, assigning DER-EMs and creating the association table, and connecting to the aggregator among others. See the docstring for each method for more details.

```
self.connect_to_gridapps()
self.load_config_from_file()
self.initialize_line_mrid()
self.establish_mrid_name_lookup_table()
```

```

self.connect_to_simulation()
self.initialize_sim_start_time()
self.initialize_sim_mrid()
self.create_objects()
self.initialize_all_der_s()
derAssignmentHandler.create_assignment_lookup_table()
derAssignmentHandler.assign_all_ders()
derIdentificationManager.initialize_association_lookup_table()
mcOutputLog.set_log_name()
goSensor.load_manual_service_file()

```

**load\_config\_from\_file()** Loads the GridAPPS-D configuration string from a file and places the parameters in a variable for later use.

**connect\_to\_gridapps()** Connects to GridAPPS-D and creates the GridAPPS session object.

**initialize\_sim\_mRID()** Retrieves the simulation mRID from the simulation object. The mRID is used to connect to messaging topics, while the object contains methods to, for example, start the simulation.

**initialize\_line\_mRID()** Retrieves the model mRID from the config parameters.

**initialize\_sim\_start\_time()** Retrieves the simulation start timestamp from the config parameters. Note: this is a setting, not the real current time.

**connect\_to\_simulation()** Connects to the GridAPPS-D simulation (as opposed to the GridAPPS-D session) and creates the simulation object.

**create\_objects()** Instantiates all non-callback classes. All objects are global to simplify arguments and facilitate decoupling. (Note: EDMCore is manually instantiated first, in the main loop function. This is part of the startup process. The callback classes need to be instantiated separately to ensure the callback methods work properly.)

**initialize\_all\_der\_s()** Calls the `[object name].initialize_der_s()` method for each DER-S listed in `mcConfiguration.ders_obj_list`.

**start\_simulation()** Performs one final initialization of the simulation start time (this fixes a bug related to our use of the logging API tricking the timekeeper into thinking it is later than it is) and calls the method to start the actual simulation.

**establish\_mrid\_name\_lookup\_table()** This currently creates two lookup dictionaries. `mrid_name_lookup_table` gets the real names of measurements for the measurement processor/logger. `cim_measurement_dict` gives a more fully fleshed out dictionary containing several parameters related to measurements that are appended to the current readings of the measurement processor.

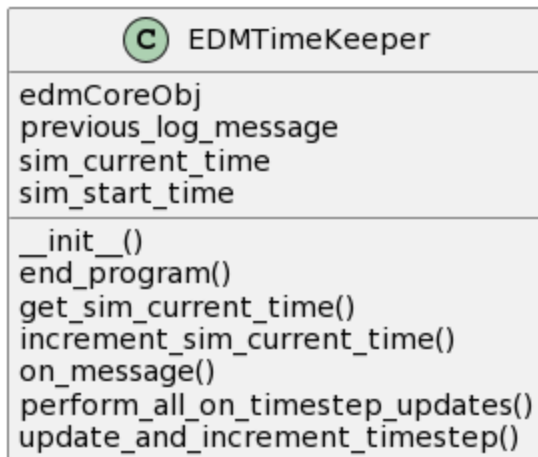
**get\_mrid\_name\_lookup\_table()** ACCESSOR METHOD: Returns the `self.mrid_name_lookup_table`.

**get\_cim\_measurement\_dict()** ACCESSOR METHOD: Returns the `self.cim_measurement_dict`.

### A.1.2 EDMTimeKeeper Class

“““CALLBACK CLASS. GridAPPS-D provides logging messages to this callback class. `on_message()` filters these down to exclude everything except simulation timestep “incrementing to...” messages and simulation ending messages. Each time an incrementation message is received from GridAPPS-D, one second has elapsed. The Timekeeper increments the time each timestep; more importantly, it also calls all methods that are intended to run continuously during simulation runtime. `perform_all_on_timestep_updates()` updates the MC once per second, including receiving DER-S inputs, updating the DER-EMs, and updating the logs.

Note: this does not include updating the grid state measurements. GridAPPS-D retrieves grid states for the measurement callbacks once every three seconds using a completely different communications pathway. As such, measurements and their processing are not handled by this class in any way.”””



### A.1.2.1 EDMTimeKeeper Attributes

**sim\_start\_time** The (from config) simulation start timecode.

**sim\_current\_time** The current timestamp. Initialized to the sim start time.

**previous\_log\_message** A buffer containing the previous log message. Necessary to fix a double incrementation glitch caused by GridAPPS-D providing the same log multiple times.

**edmCoreObj** edmCore is fed through an argument directly since it doesn't function properly as a global object.

### A.1.2.2 EDMTimeKeeper Methods

**on\_message()** CALLBACK METHOD: the “message” argument contains the full text of the Log messages provided by GridAPPS-D. This occurs for many reasons, including startup messages, errors, etc. We are only concerned with two types of messages: if the message contains the text “incrementing to,” that means one second (and thus one timestep) has elapsed, and if the message process status is “complete” or “closed”, we know the simulation is complete and the MC should close out.

**end\_program()** Submethod of `self.on_message()`. Ends the program by closing out the logs and setting the global end program flag to true, breaking the main loop.

**update\_and\_increment\_timestamp()** Submethod of `self.on_message()`. Increments the timestep only if “incrementing to” is within the log\_message; otherwise does nothing.

**increment\_sim\_current\_time()** Increments the current simulation time by 1.

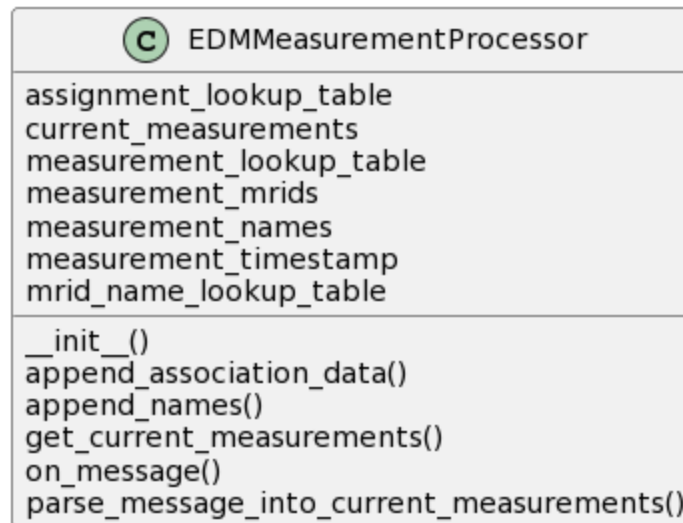
**get\_sim\_current\_time()** ACCESSOR: Returns the current simulation time. (Not real time.)

**perform\_all\_on\_timestep\_updates** ENCAPSULATION: Calls all methods that update the system each timestep (second). New processes should be added here if they need to be ongoing, I.E. once per second through the simulation. NOTE: DOES NOT INCLUDE MEASUREMENT READING/PROCESSING. Those are done once every three seconds due to the way GridAPPS-D is designed and are independent of the simulation timekeeper processes. See EDMMeasurementProcessor.

### A.1.3 EDMMeasurementProcessor Class

“““CALLBACK CLASS: once per three seconds (roughly), GridAPPS-D provides a dictionary to the `on_message()` method containing all of the simulation grid state measurements by `mRID`, including the magnitude, angle, etc. The measurement processor parses that dictionary into something more useful to the MC, draws more valuable information from the model, gets association and location data from the input branch, and appends it to the dictionary to produce something usable by the GO and the logging class.

NOTE: the API for measurements and timekeeping are completely separate. The MC as a whole is synchronized with the timekeeping class, but measurement processes are done separately. This is why logs will have repeated values: the logs are part of the MC and thus update once per second, but the grid states going IN to the logs are only updated once per three seconds.”””



#### A.1.3.1 EDMMeasurementProcessor Attributes

**measured\_timestamp** The timestamp of the most recent set of measurements as read from the GridAPPS-D message. NOTE: Currently unused, but might be useful for future log revisions.

**current\_measurements** Contains the measurements taken from the GridAPPS-D message. Written in the function `self.parse_message_into_current_measurements()`.

**current\_processed\_grid\_states** This contains the current processed grid states. Processed grid states include the measurements from `self.current_measurements` as well as DER-S to DER-EM association data and locational data.

**mrid\_name\_lookup\_table** Read from EDMCore. Used to append informative data to each measurement.

**measurement\_lookup\_table** Read from EDMCore. Used to append (different) information to each measurement.

**measurement\_mrids** Measurement dictionaries provided by GridAPPS-D use mRIDs as keys for each measurement. This contains a list of those keys and is used to replace those mRIDs with human-readable names.

**measurement\_names** A list of human-readable measurement names. See `measurement_mrids`.

**assignment\_lookup\_table** Read from DERAssignmentHandler. Used to append DER-S to DER-EM association data to each measurement for logging and troubleshooting purposes.

### A.1.3.2 EDMMeasurementProcessor Methods

**on\_message()** (Note: This is required by GridAPPS-D.) This is the “callback method” for measurements. Once per three seconds, the Simulation API provides a dictionary of dictionaries containing key-value pairs, each key being the mRID of a measurement point and each value being one or more quantities, magnitudes, or switch positions. These are given in the message argument, which can be processed by this function (and only this function) into a form usable by the rest of the script. As such, all of the actions taken each measurement timestep will be contained in this method.

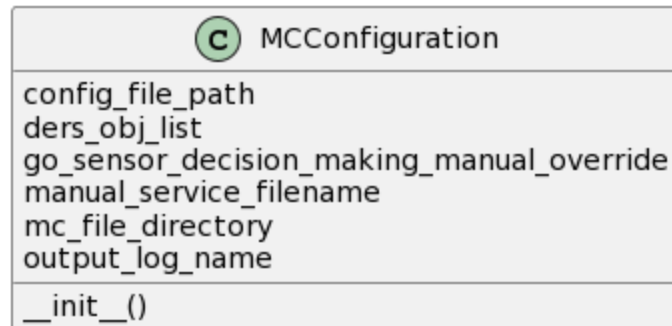
**parse\_message\_into\_current\_measurements()** This takes the contents of message (which by definition contain the newest set of measurements), parses the measurements and the timestamp, and places the values in the proper attributes. It also appends names, association data, and any other data we want to include in the measurements that have been fed through other parts of the MC.

**append\_names()** Adds important data to each measurement taken from the EDMCore lookup tables. This includes real names, equipment names, bus locations, phases, and measurement types.

**append\_association\_data()** Takes association data for each DER from the assignment lookup table and appends it to the parsed measurements by mRID. This allows locational data, for example, to be fed through the system to the GSP via the GO.

### A.1.4 MCConfiguration Class

“““Provides user configurability of the MC when appropriate. Not to be confused with the GridAPPS-D configuration process, this provides global configuration for the MC as a whole. DER-S configuration should be handled within the DER-S Class definition, and not here.”””



#### A.1.4.1 EDMMeasurementProcessor Attributes

**mc\_file\_directory** The root folder where the ME is located.

**config\_file\_path** The text file containing the GridAPPS-D configuration info.

**ders\_obj\_list** A dictionary containing the DER-S classes and objects that will be used in the current simulation. Add or comment out as appropriate for new DER-Ss or for different tests.

**go\_sensor\_decision\_making\_manual\_override** Set to True to use manual GOSensor decision making. Grid services are called by a text file rather than based on grid conditions.

**manual\_service\_filename** the XML filename of the GOSensor manual service input file. Should be in MC root.

**output\_log\_name** The name and location of the output logs. Rename before simulation with date/time, for example.

## **A.2 MC Input Classes**

### **A.2.1 RWHDEERS Class**

*“““The Resistive Water Heater DER-S. This DER-S is designed to build on prior work by the Portland State University Power Engineering Group. RWHDEERS is designed to provide a means for resistive water heaters to be modeled and simulated in the Modeling Environment.*


*The input to RWHDEERS is information from water heater emulators that are/will be provided by the GSP (via the EGoT server/client system). These emulators function over time as a resistive water heater, turning on and off based on current tank temperature, ambient losses, usage profiles, etc. These functions are handled externally to the ME, however: the end result is a series of CSV files contained in the RWHDEERS Inputs folder.*

*The ME uses these CSV files as follows. Each file is named "DER#####\_Bus####.csv". The first set of numbers is a serial number used as a 'unique identifier' for each emulated DER input. The second set is the 'locational information', in this case the Bus the DER should be located on in the model. The contents of the CSV file are a single pair of values: "P", for power, and a number corresponding to what the power should be set to. This is by agreement with the GSP designer for current testing needs; in future, the file could contain voltages, or more complex information such as usage profiles that would require modification to the RWHDEERS class to parse.*

*At the beginning of each simulation, the DERAssignmentHandler class calls the assign\_der\_s\_to\_der\_em() function for each DER-S, including RWHDEERS. This function associates each unique identifier with the mRID of a DER-EM. These DER-EMs already exist in the model and do nothing unless associated with a DER-S unit.*

*During the simulation, each time step RWHDEERS reads the CSV files for updates. The power levels for each DER are processed into a standard message format used by MCInputInterface. The association data are used to ensure each input is being sent to the proper DER-EM by MCInputInterface. Then, again on each timestep, MCInputInterface updates the DER-EMs in the model with the new power data, which is reflected in the logs.*

*In this way, changes to water heater states are converted to time-valued power changes, which are sent to RWHDEERS, processed by the MC, and written into the simulation so that grid states reflect the changes.”””*

 RWHDEERS
<code>der_em_input_request</code> <code>input_file_path</code> <code>input_identification_dict</code>
<code>__init__()</code> <code>assign_der_s_to_der_em()</code> <code>get_input_request()</code> <code>initialize_der_s()</code> <code>parse_input_file_names_for_assignment()</code> <code>update_der_em_input_request()</code>

### A.2.1.1 RWHDEERS Attributes

**der\_em\_input\_request** Contains the new DER-EM update states for this timestep, already parsed and put into list format by RWHDEERS. The list is so multiple DER-EMs can be updated per timestep.

**input\_file\_path** The folder in which the RWHDEERS input files are located.

**input\_identification\_dict** a dictionary of identification information for each DER input. The keys are the serial numbers parsed from each file name, and the values include the buses and the filename. Used during assignment, and also on time step to get the right data from the right file for each DER unique ID.

### A.2.1.2 RWHDEERS Methods

**initialize\_der\_s()** This function, with this specific name, is required in each DER-S used by the ME. The EDMCore initialization process calls this function for each DER-S activated in MCConfig to perform initialization tasks. This does not include DER-EM assignment (see `self.assign_der_s_to_der_em()`). In this case, all this function does is call the `self.parse_input_file_names_for_assignment()` function.

**assign\_der\_s\_to\_der\_em()** This function (with this specific name) is required in each DER-S used by the ME. The DERAssignmentHandler calls this function for each DER-S activated in MCConfig. The purpose of this function is to take unique identifiers from each "DER input" for a given DER-S and "associate" them with the mRIDs for DER-EMs in the model. This is done using locational data: I.E. a specific DER input should be associated with the mRID of a DER-EM on a given bus. This function does those tasks using the `input_identification_dict` generated in the initialization process.

**parse\_input\_file\_names\_for\_assignment** This function is called during the DER-S initialization process. It reads all the files in the RWHDEERS Inputs folder and parses them into an input dictionary containing the unique ID, file name, and Bus location for each. These are used during assignment and each time step to "connect the dots" between the input file and the DER-EM that represents its data.

**update\_der\_em\_input\_request()** Reads the input data from each file in the input identification dict, and puts it in a list readable by the MCInputInterface.


**get\_input\_request()** This function (with this specific name) is required in each DER-S used by the ME. Accessor function that calls for an updated input request, then returns the updated request for use by the MCInputInterface

## A.2.2 DERSHistoricalDataInput Class

“““The Historical Data DER-S. Sometimes referred to as “manual input,” this DER-S serves as a simple method to update DER-EMs manually at certain times with specific values, allowing the test engineer to write in grid states as needed by each simulation. Since DER-EMs are generic models, each historical data input could represent a single DER, groups of DERs, or even more abstract ideas such as massive power excursions.

The input is a single CSV file, contained in the DERSHistoricalData Inputs folder. This CSV is timestamped and in a specific format; after the timestamp column, columns are in pairs, with each pair representing Power and Bus for each DER-EM. The bus is used for assignment, at which point the values are associated to DER-EMs by header names.

Otherwise, it functions like any other DER-S: it has an initialization process, an assignment process, and on timestep updates.”””

 DERSHistoricalDataInput
der_em_input_request historical_data_file_path input_table list_of_ders location_lookup_dictionary
__init__() assign_der_s_to_der_em() get_input_request() initialize_der_s() open_input_file() read_input_file() update_der_em_input_request()

### A.2.2.1 DERSHistoricalDataInput Attributes

**der\_em\_input\_request** Contains the new DER-EM states for this timestep, already parsed and put into list format by the function. The list is so multiple DER-EMs can be updated per timestep.

**input\_file\_path** The folder in which the DERSHistoricalDataInput files are located.

**input\_table** The input files are in CSV format; the csv reader reads these files into a table here.

**list\_of\_ders** The DER names read from the header of the input table.

**location\_lookup\_dictionary** A dictionary associating the DER unique identifiers with the bus they should be assigned to.

### A.2.2.2 DERSHistoricalDataInput Methods

**initialize\_der\_s()** This function (with this specific name) is required in each DER-S used by the ME. The EDMCore initialization process calls this function for each DER-S activated in MCConfiguration to perform initialization tasks. This does not include DER-EM assignment (see `self.assign_der_s_to_der_em()`). In this case, all this function does is call the `self.read_input_file()` function.

**get\_input\_request()** This function (with this specific name) is required in each DER-S used by the ME. Accessor function that calls for an updated input request, then returns the updated request for use by the MCInputInterface

**assign\_DER\_S\_to\_DER\_EM()** This function (with this specific name) is required in each DER-S used by the ME. The DERAssignmentHandler calls this function for each DER-S activated in MCConfig. The purpose of this function is to take unique identifiers from each “DER input” for a given DER-S and “associate” them with the mRIDs for DER-EMs in the model. This is done using locational data: I.E. a specific DER input should be associated with the mRID of a DER-EM on a given bus.

**open\_input\_file(path)** Opens the historical data input file, reads it as a CSV file, and parses it into a list of dicts.

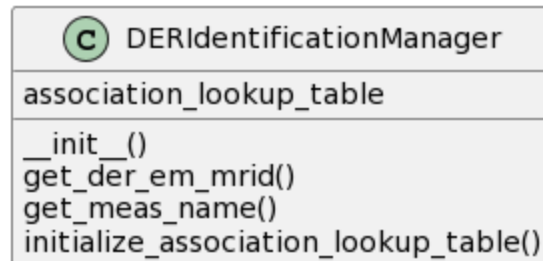
**read\_input\_file()** Reads and parses the input file. Places all the input information in `input_table`. Also, parses the CSV file to determine the names and locations of each DER: when the timestamp column is removed, odd column headers are names and even headers are their associated locations. These lists are converted to a list of dictionaries to be passed to the assignment handler (which takes the locations for each DER name and assigns a DER-EM mRID at the proper location to the name, this allows the MC to provide updated DER states to the DER-EM without requiring the inputs to know DER-EM mRIDs.)

**update\_der\_em\_input\_request()** Checks the current simulation time against the input table. If a new input exists for the current timestep, it is read, converted into an input

dictionary, and put in the current der\_input\_request (see [MCInputInterface.get\\_all\\_der\\_s\\_input\\_requests\(\)](#) )

### A.2.3 DERIdentificationManager Class

“““This class manages the input association lookup table generated by the DERSAssignmentHandler. The accessor methods allow input unique IDs to be looked up for a given DER-EM mRID, or vice versa. The table is generated during the assignment process (see DERSAssignmentHandler).”””



#### A.2.3.1 DERIdentificationManager Attributes

**association\_lookup\_table** a list of dictionaries containing association data, read from the DERSAssignmentHandler after the startup process is complete. Used to connect the unique identifiers of DER inputs (whatever form they might take) to mRIDs for their assigned DER-EMs.

#### A.2.3.2 DERIdentificationManager Methods

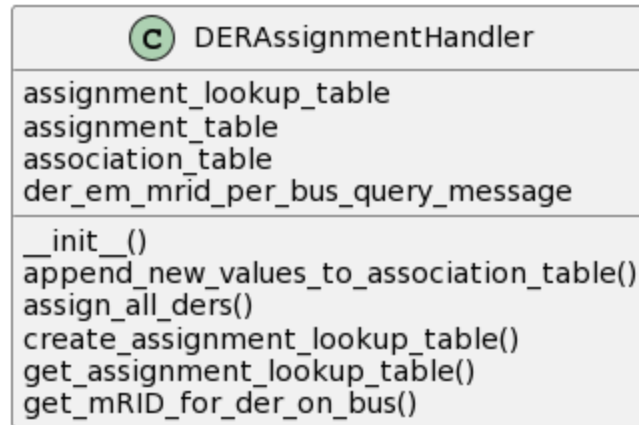
**get\_meas\_name(mRID)** ACCESSOR FUNCTION: Returns a unique identifier for a given DER-EM mRID. If none is found, the DER-EM was never assigned, and 'Unassigned' is returned instead.

**get\_DER\_EM\_mRID(name)** ACCESSOR FUNCTION: Returns the associated DER-EM control mRID for a given input unique identifier. Unlike get\_meas\_name(), if none is found that signifies a critical error with the DERSAssignmentHandler.

**initialize\_association\_lookup\_table()** Retrieves the association table from the assignment handler.

## A.2.4 DERAssignmentHandler Class

“““This class is used during the MC startup process. DER-S inputs will not know the mRIDs of DER-EMs since those are internal to the EDM, and DER inputs are external to the MC. As such, a process is required to assign each incoming DER input to an appropriate DER-EM mRID, so that its states can be updated in the model. Each DER-S DER unit requires a unique identifier (a name, a unique number, etc.) and a “location” on the grid, generally the bus it is located on. The assignment handler receives as input a list of {uniqueID:location} dictionaries, uses the location values to look up the DER-EMs on the appropriate bus, and assigns each unique identifier to an individual DER-EM. These associations are passed to the Identification Manager; during the simulation, new inputs from each unique ID are sent to the input manager, which automatically looks up the appropriate mRID for the associated DER-EM and sends the inputs there.”””



### A.2.4.1 DERAssignmentHandler Attributes

**assignment\_lookup\_table** contains a list of dictionaries containing mRID, name, and Bus of each DER-EM within the model.

**assignment\_table** a redundant `assignment_lookup_table`, used during the assignment process in order to prevent modification to the original assignment lookup table (which will still need to be used by the output branch, for example).

**association\_table** Contains association data provided by each DER-S class, for use by the DERIdentificationManager.

**der\_em\_mrid\_per\_bus\_query\_message** SPARQL Query used to gather the DER-EM info for the assignment tables from the model database.

### A.2.4.2 DERAssignmentHandler Methods

**get\_assignment\_lookup\_table()** ACCESSOR: Returns the assignment lookup table. Used in the message appendage process.

**create\_assignment\_lookup\_table()** Runs an extended SPARQL query on the database and parses it into the assignment lookup table: that is, the names and mRIDs of all DER-EMs on each bus in the current model.

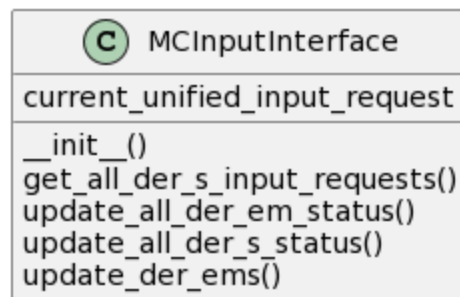
**assign\_all\_ders()** Calls the assignment process for each DER-S. Uses the DER-S list from MCConfiguration, so no additions are needed here if new DER-Ss are added.

**get\_mRID\_for\_der\_on\_bus()** For a given Bus, checks if a DER-EM exists on that bus and is available for assignment. If so, returns its mRID and removes it from the list (so a DER-EM can't be assigned twice).

**append\_new\_values\_to\_association\_table()** Used by DER-S classes to add new values to the association table during initialization.

## A.2.5 MCInputInterface Class

*“““Input interface. Receives input messages from DER-Ss, retrieves the proper DER-EM input mRIDs for each input from the Identification Manager, and delivers input messages to the EDM that update the DER-EMs with the new states.”””*



### A.2.5.1 MCInputInterface Attributes

**current\_unified\_input\_request** A list of all input requests currently being provided to the Input Interface by all active DER-Ss.

### A.2.5.2 MCInputInterface Methods

**update\_all\_der\_em\_status()** Currently, calls the `self.update_der_ems()` method. In future, may be used to call methods for different input types; a separate method may be written for voltage inputs, for instance, and called here once per timestep.

**update\_all\_der\_s\_status()** This is an encapsulation function. Once per timestep, call all methods required per DER-S to update their states.

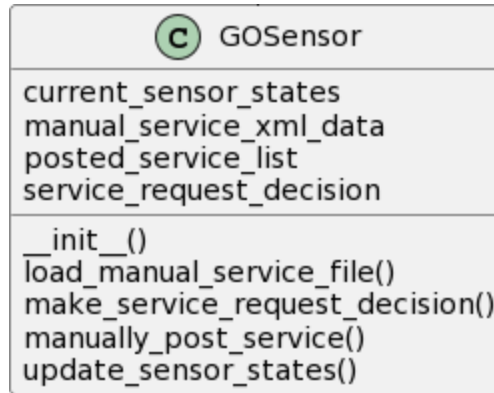
**get\_all\_DER\_S\_input\_requests()** Retrieves input requests from all DER-Ss and appends them to a unified input request.

**update\_der\_ems()** Reads each line in the unified input request and uses the GridAPPS-D library to generate EDM input messages for each one. The end result is the inputs are sent to the associated DER-EMs and the grid model is updated with the new DER states. This will be reflected in future measurements.

## A.3 MC Output Classes

### A.3.1 GOSensor Class

“““This class retrieves fully formatted grid states from the measurement processor, filters them down to necessary information, and makes determinations (automatically or manually) about grid services, whether they're required, happening satisfactorily, etc. These determinations are sent to the output API to be communicated to the DERMS.”””



#### A.3.1.1 GOSensor Attributes

**current\_sensor\_states** Grid states read into the sensor. Automatic mode only.

**posted\_service\_list** List of posted service objects. Used by both Automatic and Manual modes.

**manual\_service\_xml\_data** In Manual Mode, the data contained within the manual service xml file. To be parsed and posted service objects generated from this data.

### A.3.1.2 GOSensor Methods

**update\_sensor\_states()** Retrieves measurement data from the Measurement Processor. The measurements are organized by topological group.

**make\_service\_request\_decision()** Performs the following once per timestep.

In MANUAL MODE (override is True):

Instantiates a grid service

In AUTOMATIC MODE (override is False):


Monitors voltage and frequency, instantiates grid services, and evaluates grid services performance

**load\_manual\_service\_file()** MANUAL MODE: Reads the manually\_posted\_service\_input.xml file during MC initialization and loads it into a dictionary for later use.

**manually\_post\_service()** Called by `self.make_service_request_decision()` when in MANUAL mode. Reads the contents of the manual service dictionary, draws all relevant data points for each service, and instantiates a GOPostedService object for each one, appending the objects to a list.

### A.3.2 GOOutputInterface Class

*“““API between the MC and a DERMS. Must be customized to the needs of the DERMS. Converts determinations and feedback data to message formats the DERMS requires/can use, and delivers them..”””*

 <b>GOOutputInterface</b>
<b>current_service_requests</b>
<b>generate_service_messages()</b> <b>get_all_posted_service_requests()</b> <b>send_service_request_messages()</b>

#### A.3.2.1 GOOutputInterface Attributes

**current\_service\_requests** A list of posted services, in the form of objects of the GOPostedService class. These are the services that are being requested, or are currently being executed. Can come from either Automatic or Manual Decision making. See GOPostedService.

### A.3.2.2 GOOutputInterface Methods

**get\_all\_posted\_service\_requests()** Retrieves the service message data from each posted service (see the [GOPostedService.get\\_service\\_message\\_data\(\)](#) method for more detail). Appends the data in the proper list-of-dict format to `current_service_requests`.


Note: In the current implementation, it may seem redundant to read data from an xml file into dictionaries, package the data into an object, and extract the data back into identical dictionaries; however, this is important to ensure that the process is decoupled. A different DERMS or even a more advanced ME-GSP API might not allow for such direct input formats.

**generate\_service\_messages()** Converts the `self.current_service_requests` list of dicts into a proper xml format. Used by the xml written in `self.send_service_request_messages()`.

**send\_service\_request\_messages()** Writes the current service request messages to an xml file, which will be accessed by the GSP for its service provisioning functions.

### A.3.3 MCOutputLog Class

“““Generates CSV logs containing measurements from the measurement processor. Updates (writes a line) once per timestep.”””

 MCOutputLog
csv_dict_writer csv_file current_measurement header_mrids header_names is_first_measurement log_name mrid_name_lookup_table timestamp_array
 __init__() append_timestamps() close_out_logs() open_csv_dict_writer() open_csv_file() set_log_name() translate_header_names() update_logs() write_header() write_row()

#### A.3.3.1 MCOutputLog Attributes

**csv\_file** Contains the csv file object (see open\_csv\_file())

**log\_name** The log name, taken from MCConfiguration during initialization.

**mrid\_name\_lookup\_table** A table of mRIDs and their respective plain english names, used to create the log headers. Taken from [edmCore.get\\_mrid\\_name\\_lookup\\_table\(\)](#).

**header\_mRIDs** a list of mRIDs for each measurement point, used (invisibly) in the header to write logs.

**header\_names** the plain english versions of the header names.

**csv\_dict\_writer** The dictionary writer object, used to write the CSV logs.

**timestamp\_array** A list of all timestamps for the logs. Appended at the end of simulation.

**current\_measurement** The dictionary containing the current set of measurements.

**is\_first\_measurement** Flags functions that should only run once at the start of logging (such as opening the log files, setting up the header, etc.)

### **A.3.3.2 MCOutputLog Methods**

**update\_logs()** During the first measurement, performs housekeeping tasks like opening the file, setting the name, translating the header to something readable, and writing the header. On all subsequent measurements, it writes a row of measurements to the logs and appends a new timestamp to the timestamp array.

Note: The first timestep in the logs will be several seconds after the actual simulation start time.

**open\_csv\_file()** Opens the csv\_file object.

**open\_csv\_dict\_writer()** Opens the dict writer used to write rows. Note that the headers used are the measurement mRIDs; the plain English names are a visual effect only.

**close\_out\_logs()** Closes the log file and re-appends the timestamps.

**translate\_header\_names()** Looks up the plain english names for the headers and provides them to a dictionary for use by write\_header().

**write\_header()** Writes the log header.

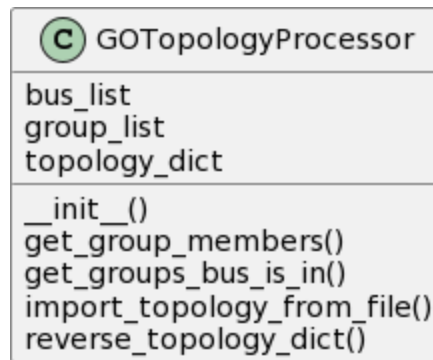
**append\_timestamps()** Uses the pandas library to append the timestamp column to the logs. This is the most convenient way to handle timekeeping while making sure to use the simulation time rather than the measurement time.

**write\_row()** Writes a row of values to the csv file.

**set\_log\_name()** Sets the log name based on the MCConfiguration settings.

### A.3.4 GOTopologyProcessor

“““ 'Topology' refers to where things are on the grid in relation to one another. In its simplest form, topology can refer to what bus each DER-EM is on. However, GOs and DERMS may view topology in more complex forms, combining buses into branches, groups, etc. More complex topologies are stored in xml files and read into the MC by this class; the XML contains each "group" and whatever buses are members of it. This class will then be able to provide that information during the assignment process or append it to measurements as needed.”””



#### A.3.4.1 GOTopologyProcessor Attributes

**topology\_dict** The table of topology information read in from the XML file

**bus\_list** A list of buses in the model

**group\_list** A list of groups in the topology table.

#### A.3.4.2 GOTopologyProcessor Methods

**reverse\_topology\_dict()** Allows groups to be references by a bus, rather than vice versa.

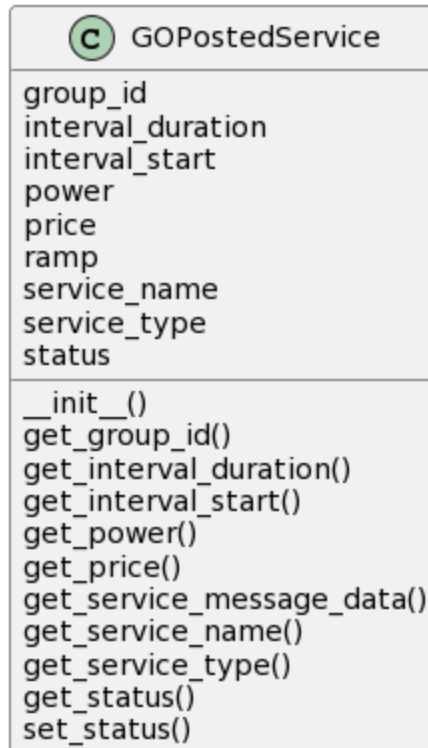
**get\_group\_members()** Returns all buses contained in a given group.

**get\_groups\_bus\_is\_in()** Returns all groups a given bus is a member of.

### A.3.5 GOPostedService

“““ This class is (currently) the only class that can get instantiated more than once per simulation. It contains service request data required to communicate with the DERMS. These data inform the DERMS of the type, location, and parameters of a single service it wants to request from the DERMS. These service requests are then 'posted' to a list in the GOSensor class, which is read by the GOOutputInterface class and processed into the communication format or protocol necessary for GO-DERMS communications (currently, an xml text file stored in the /Outputs to DERMS/ folder.)

The attributes and accessor methods are mostly self-explanatory; the major points of interest are the fact that it stores whatever data are needed by the DERMS, and that there is a function that returns these data in dictionary format.”””



#### A.3.5.1 GOPostedService Attributes

**service\_name** Name of the posted service.

**group\_id** Group identification of the posted service.

**service\_type** Type of the posted service.

**interval\_start** Start time of the posted service interval.

**interval\_duration** Duration of the posted service interval.

**power** Power requirement of the posted service.

**ramp** Ramp rate of the posted service.

**price** Price of the posted service.

**status** Boolean flag. Set to True once the service has been posted.

#### **A.3.5.2 GOPostedService Methods**

**get\_service\_name()** Returns service\_name.

**get\_group\_id()** Returns group\_id.

**get\_service\_type()** Returns service\_type.

**get\_interval\_start()** Returns interval\_start.

**get\_interval\_duration()** Returns interval\_duration.

**get\_power()** Returns power.

**get\_price()** Returns price.

**get\_status()** Returns status.

**set\_status()** Sets status to a new value.

**get\_service\_message\_data()** Returns the attribute names and values in dictionary form for use by the message wrapper (GOOutputInterface).

## Appendix B

### B.1 Example Simulation Procedure

#### B.1.1 Explanation and Configuration

The following section provides a step-by-step example of a single simulation, including preparation, configuration, the mechanisms that occur during a simulation, and the expected output. The simulation parameters are as follows:

- GridAPPS-D Simulation Configuration parameters
  - Start time: 1570041113 (*Wed Oct 02 2019 18:31:53 GMT+0000*)
  - Duration: 30 (*seconds*)
  - Model: “\_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62” (*IEEE 13 node feeder*)
- Topology
  - 1:1; i.e. each group represents a single bus, and vice versa.
- DER-Ss active
  - RWHDERS
    - Input files contained in “RWHDERS Inputs” folder.
  - DERSHistoricalDataInput
    - Input file contained in “DERSHistoricalData Inputs” folder.
- Outputs active
  - MCOutputLog
    - Saved to “Logged Grid State Data/MeasOutputLogs.csv”
  - GOOutputInterface
    - Configured to operate with GSP using XML output scheme
    - Saved to “Outputs to DERMS/OutputtoGSP.xml”
- GO Operation
  - MANUAL MODE selected
    - Inputs read from “manually\_posted\_service\_input.xml” in the MC root folder
    - Request 1:

```
<service1>
  <group_id>1</group_id>
  <service_type>"Energy"</service_type>
  <interval_start>0</interval_start>
  <interval_duration>0</interval_duration>
  <power>1000</power>
  <ramp>0</ramp>
  <price>0</price>
  <start_time>1570041123</start_time>
</service1>
```

- Request 2:

```

<service2>
  <group_id>2</group_id>
  <service_type>"Energy"</service_type>
  <interval_start>0</interval_start>
  <interval_duration>0</interval_duration>
  <power>1000</power>
  <ramp>0</ramp>
  <price>0</price>
  <start_time>1570041128</start_time>
</service2>

```

- Purpose of simulation
  - Functional test: ensure inputs from DERSHistoricalDataInput are being reflected in the output logs
  - **Note:** GSP will not be operating during this simulation. Mock inputs showing no effect will be used and outputs to GSP will be ignored. However, the processes will be explained all the same.

## B.1.2 MC Installation

The following is a step-by-step process on installing and configuring the MC and CIMHub scripts.

1. If not already done, install Python 3.
2. If using Windows, install Ubuntu.<sup>7</sup> If using Linux, this step shouldn't be required.<sup>8</sup>
3. Install Docker.<sup>9</sup>
4. Install GridAPPS-D.<sup>10</sup>
5. Clone the doe-egot-me GitHub repository<sup>11</sup> to your local system.
6. Clone the CIMHub<sup>12</sup> and Powergrid-Models<sup>13</sup> repositories to your local system.
7. Make the following edits to ModelController.py:
  - a. In the `MCConfiguration.__init__()` method, edit "`self.mc_file_directory`" with the correct file path for your computer. Verify other path and filename attributes are correct.
  - b. Verify the path attributes are correct in the `__init__()` methods of all DER-S classes.
8. Navigate to the `/DERScripts/` directory and make the following changes to `envvars.sh`
  - a. Edit the `SRC_PATH` variable with the correct path for the `/Powergrid-Models/platform/` folder you cloned.

<sup>7</sup> <https://ubuntu.com/>

<sup>8</sup> The ME was designed for cross-platform support and GridAPPS-D is executed in Linux while the MC is executed directly from the command line. Mac support via a Linux command line is expected, but has not been tested.

<sup>9</sup> <https://www.docker.com/products/docker-desktop/>

<sup>10</sup> [https://gridappsd.readthedocs.io/en/master/installing\\_gridappsd/index.html](https://gridappsd.readthedocs.io/en/master/installing_gridappsd/index.html)

<sup>11</sup> <https://github.com/PortlandStatePowerLab/doe-egot-me>

<sup>12</sup> <https://github.com/GRIDAPPSD/CIMHub>

<sup>13</sup> <https://github.com/GRIDAPPSD/Powergrid-Models>

- b. Edit cimhubconfig.json, ensure it reads:

```
{  
  "blazegraph_url": "http://localhost:8889/bigdata/namespace/kb/sparql",  
  "cim_ns": "<http://iec.ch/TC57/CIM100#"
```

### B.1.3 Pre-Simulation Configuration

1. Start GridAPPS-D
  - a. From the command line, navigate to the GridAPPS-D docker container folder and type `./run.sh` to start the container. The most recent version of GridAPPS-D will be downloaded as necessary, along with the updated blazegraph database.
    - i. **Important note:** Any new version of GridAPPS-D will overwrite the models in the database. The DER-EM addition process will need to be performed after any update. See below for the process to do so.
  - b. Within the container, type `./run-gridappsd.sh` to start the GridAPPS-D program.
    - i. **Note:** If necessary, press CTRL-C to exit a running GridAPPS-D program. Type `exit` to leave the gridappsd docker container. Type `./stop.sh -c` to close the docker container.
2. Add DER-EMs to the grid model
  - a. Configure the DER addition scripts as follows:
    - i. In the DERScripts folder, verify, create or modify and equivalent to `EGoT13_der.txt` with the required number and labelplate info for each DER-EM required as well as the mRID for the proper model.
    - ii. Verify, modify, or create as needed `EGoT13_orig_der.txt` or an equivalent containing information on any DER that is included in the model by default but should be removed; for example, in the IEEE 13 node feeder, by default the model contains a house and a school, which should be removed.
    - iii. Modify (or verify) each script within the `Initialise_DER_EM.sh` script for the proper feeder model:
      1. `drop_orig_der.sh`: ensure the proper text file from (ii) is entered.
      2. `drop_der.sh`: ensure that an equivalent to `EGoT13_der_uuid.txt` is entered. This file contains the DER-EM uuids generated by a previous DER-EM addition script execution; if this is your first time adding DERs to a new model, this file will not yet exist and thus can be ignored.
      3. `insert_der.sh`: ensure the file from (i) is entered.

4. drop\_all\_measurements.sh: uncomment (or, for a new model, add) the line including the mRID for the desired feeder model.
  5. list\_all\_measurements.sh: uncomment (or, for a new model, add) the line including the mRID for the desired feeder model.
  6. insert\_all\_measurements.sh: uncomment (or, for a new model, add) lines corresponding to the name of the desired feeder model. If this is the first time adding DER-EMs to a new model, this name is taken from column 4 of the line in list\_all\_measurements.sh.
- b. Execute the DER-EM addition scripts as follows.
- i. From the MC root folder, execute "Initialise\_DER\_EMs.bat" or, from the DERScripts folder, execute "Initialise\_DER\_EMs.sh". The following steps occur:
    1. drop\_orig\_der.sh removes "original" DERs from the model; that is, DERs contained in the default models provided by GridAPPS-D, but which the TE wished to be removed by default.
    2. drop\_der.sh removes any DER-EMs added to the model by previous DER-EM addition runs. At this point, the model is a "blank slate" containing no DER-EMs.
    3. insert\_der.sh adds DER-EMs to the model as listed in EGoT13\_der.txt (or equivalent). It also automatically generates uuids for the DER-EMs and places them in the "EGoT13\_der\_uuid.txt" (or equivalent) file. The model now contains DER-EMs and their controls; however, there are no associated measurement points yet.
    4. drop\_all\_measurements.sh removes any measurements contained in the files in the Meas folder associated with the selected grid model. As (2) above, this gives us a "blank slate" and prevents duplicate measurement points when the DER-EM addition script is run more than once.
    5. list\_all\_measurements.sh generates measurement point mRIDs for the selected grid model, collects them in files and adds them to the Meas folder.
    6. insert\_all\_measurements takes the mRIDs generated in (5) and inserts them into the grid model for the associated DER-EMs.
  - c. At this point, all DER-EMs are added, excess DERs have been removed, and everything in the model has all necessary mRIDs. The grid model is ready for use in the MC; if no GridAPPS-D updates or model modifications are required, this process needs only be run once.
3. Update GridAPPS-D simulation configuration for the simulation parameters (Section B.1.1):
- a. In the Configuration folder, make edits or verify the following in Config.txt:

- i. The Geographical Region, SubGeographicalRegion, and Line mRIDs for the feeder model in use.
  - ii. start\_time
  - iii. duration
- b. In the Configuration folder, verify or edit topology.xml with the correct topology
  - i. For this example, each group should hold a single bus, and each bus should be a member of only one group. An example of the syntax is:

```
<group name= "group-1" >
    <bus name="650" />
</group>
```

#### 4. Configure and enable each DER-S

- a. DERSHistoricalDataInput:
  - i. Create an input log. The first column must contain UNIX timestamps. Each pair of columns after this represents the input to a single DER-EM, with the unique identifier being the header of the first column in the pair. For each column pair, the first column must contain Watt values, and the second column must contain the bus the DER should be assigned to (and therefore will be the same value in every row). This log must be saved as a CSV file in the “DERSHistoricalData Inputs” folder.
  - ii. Modify ModelController.py as follows
    - 1. DERSHistoricalDataInput class:
      - a. Modify or verify that the `__init__()` method assignment of `self.historical_data_file_path` attribute is set to the input file (see i.)
    - 2. MCCConfiguration class
      - a. Modify or verify that the `__init__()` method assignment of `self.ders_obj_list` includes the DERSHistoricalDataInput object by adding or uncommenting the following line in the dictionary:

```
'DERSHistoricalDataInput': 'dersHistoricalDataInput',
```

#### b. RWHDERS

- i. Create or verify mock input files with the following format:
  - 1. The filename for each must be in “DER#####\_Bus\$\$\$ .csv” with “#####” form representing a unique identifier and “\$\$\$” is the bus (locational identifier).
  - 2. The contents of each must be “P, #####” with P standing for “power” and “#####” representing the Watt value of power consumption.
  - 3. Note: Due to the parameters of this simulation, the GSP will not be active. As such, these files will not change

throughout the simulation; the initial values of the mock inputs will be used through the entire simulation.

- ii. Modify ModelController.py as follows
  1. RWHDERS class:
    - a. Modify or verify that the `__init__()` method assignment of `self.input_file_path` attribute is set to the directory containing the mock inputs (see i.)
  2. MCConfiguration class
    - a. Modify or verify that the `__init__()` method assignment of `self.ders_obj_list` includes the RWHDERS object by adding or uncommenting the following line in the dictionary:

```
'RWHDERS': 'rwhDERS'
```

5. Configure the GO and outputs
  - a. Set GO to MANUAL operation mode:
    - i. Modify or verify ModelController.py as follows:
      1. Modify or verify the `MCConfiguration.__init__()` method  
`self.go_sensor_decision_making_manual_override` attribute is set to `"True"`.
    - ii. Ensure a properly formatted "manually\_posted\_service\_input.xml" file exists in the MC root directory
      1. Note: In this simulation, grid service requests will be generated and posted; however, without a GSP to receive them, they'll simply sit in the "Output to DERMS" directory without being used. Since they'll still be generated, a proper input file is required.
  - b. No configuration is required for the MCOutputLog; the output file name and path can be set by modifying the `MCConfiguration.output_log_name` attribute.
  - c. No configuration is required for the GOOutputInterface; the output file name and path can be set by modifying the `GOOutputInterface.send_service_request_messages()` method.
6. The MC is now ready. The TE may begin a simulation.

### B.1.4 Simulation execution

1. If not already completed, start GridAPPS-D and configure the system (see section B.1.3)
2. Run the ModelController.py script from an IDE or the command line.
  - a. No further inputs should be required for the duration of the simulation. Proper operation should be verified via the terminal.

- b. **IMPORTANT NOTE:** A glitch in GridAPPS-D causes very frequent instances of GridAPPS-D freezing during the simulation startup process. If the simulation doesn't seem to be starting in short order, press CTRL-C in the terminal to close GridAPPS-D, type `./run-gridappsd.sh` to re-run the program, and attempt to rerun the ModelController.py script.
3. When the simulation is complete, measurement logs will be generated and placed in the "Logged Grid States Data" directory. These logs can be used to verify DER-EMs operated per the input data sent to DERSHistoricalDataInput. The functional test is complete.

## **B.2 PSU IEEE 13-Node Feeder Simulation Example**

In its current state, the PSU IEEE 13-Node Feeder incorporates  $\approx 1000$  houses; each house includes a DER. The house objects within the model can only be changed by editing the OpenDSS file. The number of added DERs, however, can be configured by modifying the "EGoT13\_der.txt" file, as shown in step B.1.3.

This section is divided in two parts: Modifying PSU IEEE 13-node feeder and Pre-simulation Configuration. The Modifying PSU IEEE-13 Node feeder section provides insights into how to edit the PSU IEEE-13 Node Feeder file, if needed, export the required CIM XML file, and upload the CIM XML file to the Blazegraph database. The Pre-simulation Configuration shows how configure the MC to run a simulation. Note that this section does not discuss validating the OpenDSS model. The validation process is explained in the CIMHub documentation.<sup>14</sup>

### **B.2.1 Modifying the PSU IEEE 13-Node feeder**

#### **B.2.1.1 Dependencies**

1. Install OpenDSS (required)<sup>15</sup>
2. Install GridLAB-D<sup>16</sup> (if validation is needed)

**Note:** Ensure the OpenDSS version is 1.2.11. The latest version of OpenDSS exports the CIM XML objects mRIDs without prepended underscores<sup>17</sup>, which is not compatible with the current GridAPPS-D version. Any OpenDSS version above the 1.2.11 will result in an empty measurement files and, therefore, no measurements will be captured during the simulation.

#### **B.2.1.2 Feeder Configuration**

The PSU Feeder OpenDSS files are located within the "*DERScripts*" folder inside the root MC directory. Changing the configuration of the feeder, the rated values, or the names associated with any object within the feeder will require generating a new CIM XML file. This CIM XML file can be generated using OpenDSS to reflect the changes made to the feeder. The CIM XML file will then be uploaded to the Blazegraph database.

---

<sup>14</sup> <https://cimhub.readthedocs.io/en/latest/Tutorial.html#ieee-123-bus-base-case>

<sup>15</sup> <https://sourceforge.net/projects/electricdss/>

<sup>16</sup> <https://www.gridlabd.org/downloads.stm>

<sup>17</sup> [Our discussion in OpenDSS Forum regarding the prepended underscores in CIM XML file.](#)

The following steps explain how to edit the feeder model and generate a new XML CIM file.

1. Install the OpenDSS software.
2. Follow the instructions in section B.1.2 to install and configure the MC and the CIMHub scripts.
3. Navigate to the /DERScripts/dss\_files/ folder
4. The PSU feeder model is in a file named “Master.dss”
5. Edit the “Master.dss” file as needed and save the changes.
6. Within the same folder, edit the cim\_test.dss file.
  - a. In the first line, change the path to point to the modified Master.dss file.
7. Using OpenDSS, run the cim\_test.dss file.
8. If configured correctly, step 7 will execute without errors and generate a CIM XML file named “Master.xml”.

## B.2.2 Pre-Simulation Configuration

The “Master.xml” file now holds the model mRID as well as the mRIDs for each object within the feeder. A subset of these mRIDs will be used in the DER addition scripts. This subsection follows the steps previously described in section B.1.3. However, minor required modifications to the DER-EM addition scripts will be highlighted below.

1. Start GridAPPS-D:
  - a. Follow the instructions described in section B.1.3, steps 2.a and 2.b
2. Adding DER-EMs to the PSU feeder model:
  - a. Follow the instructions in section B.1.3 steps 2.a.i.
  - b. Since the PSU feeder is initialized without DER-EMs, step 2.a.ii is not needed.
3. Uploading the Model to the GridAPPS-D database:
  - a. Navigate to the /DERScripts/psu\_feeder\_files/ folder.
  - b. Using Python3, simply run “python3 upload\_model.py” script. By default, this script will upload the newly created grid model to the Blazegraph database **and** remove all other existing models within the database.
  - c. If the TE wants to keep the other models in the Blazegraph database:
    - i. Edit the “upload\_model.py” Python script.
    - ii. In the main function, comment out “*remove\_all\_feeders()*” method.
4. Extracting the model mRIDs:
  - a. Navigate to the /DERScripts/psu\_feeder\_files/ folder.
  - b. Using Python3, run “python3 extract\_mrids.py” script. This script will send a query to Powergrid Model API to retrieve the information of all available models in the GridAPPS-D database. The query response is filtered to print out the following dictionary:

```
{'modelId': '_89869331-6171-421C-95D8-55D61BCA706D',  
  'modelName': 'psu_13_node_feeder',  
  'regionId': '_9E64579D-216C-4731-B05B-6EE7FE68DB46',  
  'regionName': 'Oregon',
```

```
'stationId': '_2B50FE7F-ADDE-41C5-A76A-52CE0E5E4535',
'stationName': 'Fictitious',
'subRegionId': '_2498B425-DE51-4692-8D20-9D8D16CB23CC',
'subRegionName': 'Portland'}
```

**Figure B.1:** PSU IEEE 13-Node Feeder information

5. Edit the “Initialize\_DER\_EMs.sh” or “Initialize\_DER\_EMs.bat” scripts:
  - a. Follow the instructions in section B.1.3, steps 2.a.i, 2.a.iii.3, 2.a.iii.5, and 2.a.iii.6.
    - i. The `'modelId'` from figure B.1 is used in steps B.1.3.2.a.i and B.1.3.2.a.iii.5.
    - ii. The `'modelName'` from figure B.1 is used in step B.1.3.2.a.iii.5.
  - b. The other initialization scripts are skipped because the PSU Feeder does not incorporate any DER-EMs. These DER-EMs are added using the EGoT13\_der.txt file, as shown in the in step B.2.2.2.
6. Executing the DER-EM addition scripts:
  - a. Navigate to the /DERScripts/ folder.
  - b. Edit the “Initialise\_DER\_EMs.sh” files:
    - i. Comment out the drop\_orig\_der.sh line.
    - ii. Comment out the drop\_der.sh line.
    - iii. Comment out the drop\_all\_measurements.sh line.
  - c. Execute the “Initialise\_DER\_EMs.bat” from the MC root directory or “Initialise\_DER\_EMs.sh” files from the DERScripts folder.
7. Updating the GridAPPS-D simulation configuration:
  - a. Within the MC root directory, /Configuration/ folder, modify the Config.txt file as follows:
    - i. The `'regionId'`, `'subRegionId'`, `'modelId'`, and `'modelName'` are replaced with Geographical Region, SubGeographical Region, Line mRID, and simulation name, respectively.
  - b. The PSU feeder topology is located within the /Configuration/ folder, in a file named “psu\_feeder\_topology.xml”. Each group incorporates a feeder and several segments, transformers, and service points. The TE may change the topology as they wish. However, ensure that the names of the objects within the topology file match the names of the objects within the feeder file (i.e. Master.xml).
8. Configure and enable each DER-S
  - a. The configuration of the DER-S class are identical to the instructions in section B.1.3, step 4.
9. Configure the GO and the outputs
  - a. If the GO is set in MANUAL operation mode, follow the instructions in section B.1.3, step 5.
  - b. If the GO is set in Automatic operation mode, no configurations will be required.
10. The TE may now run the simulation.

### B.2.3 Simulation execution

Follow instructions in section B.14

## B.3 Simulation Process Summary

The following is a simplified summary of the inner workings and information exchanges of the ME simulation configured in (B.1 and B.2). This is intended to be a high level overview of operations, and not a full algorithmic description of the system.

### B.3.1 Simulation Top Level Process

1. The Test Engineer, having previously configured the system, executes “ModelController.py”.
2. Prior to major class instantiation, the MC performs the following tasks:
  - a. The “end\_program” flag is initialized to “False”.
  - b. Python libraries are imported.
  - c. Classes and functions are defined.
3. The MC runs “Program Execution”, including the “main loop”:
  - a. MCConfiguration is instantiated as a global object.
  - b. EDMCore is instantiated as a global object.
  - c. The following processes are called:
    - i. the EDMCore simulation startup processes (See B.3.1.1)
    - ii. the callback class instantiation (See B.3.1.2)
  - d. The script enters the “main loop.” In this state, all system functions are handled by the callback classes and their respective function calls (See B.3.2). Meanwhile, a while loop verifies that the end\_program variable is “False” and, if so, pauses itself briefly. If the end\_program variable is set to “True”, quit() is called to end the program.

#### B.3.1.1 EDMCore Simulation Startup Processes

1. edmCore is instantiated
  - a. `__init__()` is called, creating all attributes. No significant processes occur.
2. `edmCore.sim_start_up_process()` is called.
  - a. `self.connect_to_gridapps()` is called.
    - i. A gridappsd library function is called, connecting the MC to the GridAPPS-D program (not simulation) by assigning an object to the `self.gapps_session` attribute.
  - b. `self.load_config_from_file()` is called.
    - i. The GridAPPS-D config parameters are read in and parsed from the Config.txt file.
  - c. `self.initialize_line_mrid()` is called.
    - i. The “line” (or model) mRID is parsed from the config and assigned to the proper attribute.
  - d. `self.establish_mrid_name_lookup_table()` is called.
    - i. Two queries are sent to the model in the database via the `self.gapps_session` object. These queries return dictionaries containing the mRID-Name Lookup Table and the CIM Measurement Dictionary for later use by the logger.

- e. `self.connect_to_simulation()` is called.
  - i. This establishes the simulation in GridAPPS-D using the `gapps_session` object and the `config_parameters` read in. Generates an object assigned to `self.sim_session`.
- f. `self.initialize_sim_start_time()` is called.
  - i. The simulation start time attribute is initialized using the start time from the config.
- g. `self.initialize_sim_mrid()` is called.
  - i. Retrieves the mRID for the simulation, called from the `self.sim_session` object. Places it in the `self.sim_mrid` attribute.
- h. `self.create_objects()` is called.
  - i. Calls global instances of the following non-callback classes:
    1. `MCOutputLog`
    2. `MCInputInterface`
    3. `DERSHistoricalDataInput`
      - a. On construction, sets the `DERSHistoricalDataInput.historical_data_file_path` attribute based on the `MCConfiguration` attribute and user configuration.
    4. `RWHDERS`
      - a. On construction, sets the `RWHDERS.input_file_path` attribute based on the `MCConfiguration` attribute and user configuration.
    5. `DERAssignmentHandler`
    6. `DERIdentificationManager`
    7. `GOSensor`
    8. `GOOutputInterface`
- i. `self.initialize_all_der_s()` is called.
  - i. For each DER-S considered “active” (that is, enabled in `mcConfiguration`), the `[object name].intialize_der_s()` method of the DER-S is called. This method with this name must be included in every DER-S, customized to its needs, and performs the following:
    1. Connects to or reads in and input data
    2. Using input data, generates a list of DERs to be assigned; each dictionary in said list contains the unique identifier and locational identifier of each DER to be assigned.
- j. `derAssignmentHandler.create_assignment_lookup_table()` is called.
  - i. This queries the model database and produces a list of DER-EMs by locational identifier and mRID.
- k. `derAssignmentHandler.assign_all_ders()` is called.
  - i. This method connects the previous two steps. In the first step, a table of DER Inputs by unique identifier and location was generated; in the second step, a table of DER-EMs by location and mRID was generated. In this step, the assignment handler

steps through each DER Input for each DER-S, one by one. For each DER Input, the locational identifier is read, and the assignment handler attempts to locate an unassigned DER-EM on that location. If successful, it returns a dictionary associating the unique identifier with the mRID, which is added to a list: the association table. This continues until the unique identifier of each DER Input is assigned to a DER-EM mRID; or, if not enough DER-EMs exist at the proper location, the system exits with an explicit error informing the TE.

- l. `derIdentificationManager.initialize_association_lookup_table()` is called.
  - i. After the Assignment Manager has completed the assignment task, the association manager is moved to the Identification Manager, which has methods allowing mRIDs to be referenced by unique identifiers and vice versa.
- m. `mcOutputLog.set_log_name()` is called.
  - i. Sets the log name. This method can either assign the attribute directly or be modified to generate custom log names based on system time, for example. Hence the method call.
- n. `goSensor.load_manual_service_file()` is called.
  - i. Loads the XML file containing service requests to be called in Manual Mode.

### B.3.1.2 Callback Class Instantiation

1. Global function `instantiate_callback_classes()` is called.
  - a. Global callback object `edmMeasurementProcessor` is instantiated.
  - b. Global callback object `edmTimekeeper` is instantiated.
    - i. On construction, the `edmTimekeeper.sim_start_time` and `edmTimekeeper.sim_current_time` attributes are set to the start time from the Config.txt file, ensuring that timekeeping starts at the proper time.

## B.3.2 Callback Processes

After the simulation start up processes have been completed, all script functions are handled within the callback classes. EDMTimekeeper handles all functions that are intended to call frequently, and updates once per timestep, or once per second. EDMMeasurementProcessor is dedicated to receiving measurements from the simulation, parsing them into the proper form, and making them available to the GO and logger; it updates less frequently around once per three seconds.

### B.3.2.1 On-Timestep Functions

1. Frequently each second, log messages are sent from the GridAPPS-D simulation to the `edmTimekeeper` object. These messages are sent for many reasons, including errors, system changes, and (internal) simulation timestep incrementation. This invokes `edmTimekeeper.on_message()`, performing the following tasks.

- a. The GridAPPS-D log message is parsed. If the message indicates that the process is “COMPLETED” or “CLOSED”, `self.end_program()` is called; this closes out the log file and ends the program. If the log message contains the words “incrementing to”, a timestep has occurred and on-timestep functions are called (see b.) Otherwise, the message is disregarded.
- b. If an incrementation message has been detected, the incrementation method first detects if the log message is a repeat of the last received message. GridAPPS-D often sends the same log message multiple times; this check ensures duplicate messages don’t invoke multiple on-timestep processes or incrementations.
- c. `self.increment_sim_current_time()` is called.
  - i. This increments the `edmTimekeeper.sim_current_time` by 1.
- d. `self.perform_all_on_timestep_updates()` is called. This is the encapsulation method that performs all of the system processes that should be done frequently. They include the following:
  - i. The edmCore current time is matched to the edmTimekeeper current time.
  - ii. `mcInputInterface.update_all_der_s_status()` is called.
    1. This method generates the “unified input request” by calling the `[object name].get_input_request()` method of each DER-S and appending the results to a list. Said results are a list of dictionaries for each DER-S containing the unique identifiers of DER inputs as keys, and power in Watts as values. These values are taken from the most up-to-date DER inputs and represent the state the DER-EMs should be updated to. In short, the unified input request contains all the DER unique IDs to be updated, and the power values they should be updated to.
  - iii. `mcInputInterface.update_all_der_em_status()` is called.
    1. This method reads each line of the unified input request and replaces each unique identifier with the mRID of the DER-EM that DER input has been assigned to (see B.2.1) by referencing the Association Lookup Table in the `derIdentificationManager`. The result is a new unified input request: this table associates mRIDs with power values. Then, for each item in the unified input request, a message is generated in the proper format containing the target DER-EM mRID and the power; these messages are sent to the EDM via the `edmCore.gapps_session` object. This causes the DER-EM power values to be adjusted to the new values.
  - iv. `mcOutputLog.update_logs()` is called.
    1. If at least one measurement has been parsed by the `edmMeasurementProcessor`, this method updates the logs. The first time this occurs, the log file is created, the log headers are translated from mRIDs to human-readable

names via the mRID-Name Lookup Table in edmCore, the header is written, and a flag is set indicating the logs are ready to be written. If this is not the first measurement, then a line of the logs is written and timestamped using the edmTimekeeper current time.

- v. `goSensor.make_service_request_decision()` is called.
  - 1. This method is the same for Automatic and Manual GO operations and uses the configuration flag to determine how to proceed. In this case, Manual mode has been selected; so, it calls the `goSensor.manually_post_service()` method.
  - 2. `goSensor.manually_post_service()` checks the `manually_posted_service_input.xml` file contents (loaded into a dictionary in the startup process) to see if any services should be posted in the current timestep. If so, it instantiates a `GOPostedService` object whose attributes are initialized to the values from the input dictionary, and appends the object to the `goSensor.posted_service_list` to be read by the `GOOutputInterface`.
- vi. `goOutputInterface.get_all_posted_service_requests()` is called.
  - 1. The first step in the GO-DERMS communication process. Each `goPostedService` object on the `goSensor.posted_service_list` is scanned; if the “status” flag indicates it hasn’t been packaged yet, the service request parameters are pulled from the object attributes and placed in a standardized dictionary. This list of dictionaries is used to generate messages in the next step.
- vii. `goOutputInterface.send_service_request_messages()` is called.
  - 1. The second and final step in the GO-DERMS communication process. The list of dictionaries in `goOutputInterface.current_service_requests` is converted to XML format and written to (in this case) an output file to be read by the GSP. In this case, the GSP is inactive so nothing is done with the file; in other simulations, the GSP would read and parse that file to determine when and how to dispatch resources to fulfill the request.

### B.3.2.2 On-Measurement Functions

- 1. Once every three seconds, a message is sent from the EDM to the `edmMeasurementProcessor` object. This message is a very large dictionary of dictionaries containing grid state measurements for every item in the grid model. Each measurement is a dictionary containing multiple key/value pairs for mRID,

measurement value, measurement angle, etc. Each set of measurements is timestamped, and each measurement is referenced by mRID; however, amplifying data such as readable names, measurement types, phase, bus location, etc. are not included and must be added by the measurement processor. Each measurement message invokes the `edmMeasurementProcessor.on_message()` method, performing the following tasks:

- a. The message is parsed for the current measurements, which are placed in `self.current_measurements`.
- b. The message is parsed for the measurement timestamp, which is placed in `self.measurement_timestamp`.
- c. `self.append_names()` is called.
  - i. This method accesses the MRID-Name Lookup Table and the Measurement Lookup Table from `edmCore`, and retrieves the measurement mRIDs from the current measurement message. Using those mRIDs, the readable names are pulled from the MRID-Name Lookup Table and added to the individual measurement dictionaries. It then pulls the following values from the Measurement Lookup Table: Measurement Name, Conducting Equipment Name, Bus, Phases, and MeasType and adds these data to the dictionary for each measurement.
- d. `self.append_association_data()` is called.
  - i. This method appends the association data from the `derAssignmentHandler` lookup table to each measurement; this associates the measurement with the piece of equipment as well as its operating mRID and unique identifiers. This information is useful to reference DER Inputs against resultant grid state changes reflected in the log outputs.

## ***B.4 DER-S Design***

Each DER-S serves as an interface between some form of DER input and the MC. These data can come from systems, scripts, physical modules, sensors, or any other input format imaginable. This makes it impossible to develop a single generic interface between the inputs and the MC: a new interface should be developed for each new type of input.

The MC is structured to make the development and integration of new inputs relatively user-friendly while allowing for complete flexibility in how the DER-S reads and processes inputs. The following section describes the architecture of a “generic” DER-S including mandatory components, as well as an example of how one DER-S was developed.

### **B.4.1 Overview**

A DER-S is, ultimately, a communication interface. Because of this, it can be described in terms of Inputs, Processing, and Output. Each timestep, Input functions retrieve updated DER states from the DER Input. For example, this could be the current sensor states stored in an input buffer, or a line of data read from a log file containing DER

power levels at the given time. Then, Processing functions convert that operating data into something useful to the MC. If a DER Input does not provide power directly, but does provide operating states and nameplate data, these functions can calculate the proper power level at a given time. Finally, Output functions generate a standardized list of dictionaries containing each DER Input Unique Identifier as key and a numeric power as value, and provide them to the MCInputInterface to be delivered to DER-EM controllers.

One complicating factor in DER-S design is integrating the DER-S with the MC once it has been developed. Each DER-S has interactions with the DERAssignmentHandler and MCInputInterface classes; manually adding or removing functions for each DER-S to both of said classes each time a simulation is reconfigured would be complicated, particularly if many DER-Ss are being used. To simplify integration, certain requirements have been put in place with respect to how DER-S functions are named and called by the system. These are addressed below.

## B.4.2 Mandatory Methods

The following methods are required to be included in all DER-S classes. These methods *must* have the specific name given. This is because the MC puts each DER-S object in a list in the MCConfiguration object; when a certain process is required, the MC iterates through the list and calls the method with that name for each member in the list. For example, every timestep the MCInputInterface updates its inputs by calling the `[object_name].get_input_request()` method of every active DER-S.

1. `self.initialize_der_s()` performs any functionality required during the simulation startup process *before* assigning DER-Ss to DER-EMs. Typically this would include making connections to external systems, reading input files into the DER-S, and/or making lists of DER Input unique identifiers and locational identifiers for the assignment process.
2. `self.assign_der_s_to_der_em()` iterates through the list of DER Inputs/locational identifiers generated by `self.initialize_der_s()`, and for each item, retrieves a DER-EM mRID for assignment based on the locational identifier. This mRID is placed in a dictionary with a unique identifier as key and mRID as value; this dictionary is appended to the association table via the `DERAssignmentHandler.append_new_values_to_association_table()` method. The process is repeated for each DER Input handled by the DER-S, resulting in each DER Input being assigned to a DER-EM.
3. `self.get_input_request()` typically performs two steps. The first step is to call a method, usually named `self.update_der_em_input_request()` (though this name is not mandatory). This method contains the Processing function for the DER-S: it performs whatever steps are necessary in whatever order is required to retrieve updated DER Input states, process them into updated power levels for the current timestep, and places them in dictionaries with the DER Input unique identifier as key, and the numeric power level as value. This dictionary is appended to a list of dictionaries for each DER Input; this list is normally (but not required to be) named `self.der_em_input_request`. The second step in the

`self.get_input_request()` process is mandatory: the method returns the attribute containing the list of updated DER input requests.

These are the mandatory methods; other methods may be written and called by the mandatory methods or other MC components (e.g. the EDTimeKeeper) to provide more advanced functionality.

### B.4.3 Installing a New DER-S

Once a new DER-S class is written, follow these steps to integrate it into the Model Controller.

1. **Include Class Definition:** Include the class within the ModelController.py script, either by adding the class definition to the “Class Definitions” section, or including it as a module from another file.
2. **Define Class and Instantiate Object:** Within ModelController.py, navigate to the `EDMCore.create_objects()` method definition. Declare the object as a global variable, and instantiate the object with the argument (mcConfiguration). For example, for a DER-S named “DERSEExample”, use the following syntax:

```
global dersExample
dersExample = DERSEExample(mcConfiguration)
```

### B.4.4 Configuring and Activating a DER-S

Once the installation process is complete (see section B.3.3) the DER-S is ready to be used. However, the mandatory function calls will not occur for a DER-S unless it is activated by performing the following step:

- In ModelController.py, edit the `MCConfiguration.ders_obj_list` attribute and add a new dictionary to the list. Each dictionary has the following format: `{'CLASS_name': 'object_name'}`. For example, for a DER-S named DERSEExample instantiated to an object names dersExample, add a dictionary to the list that reads `{'DERSEExample': 'dersExample'}`.
- If more than one DER-S is active in the list at a time, ensure that the proper python dictionary format is being maintained including commas. For example:

```
self.ders_obj_list = {
    'DERS_1' : 'ders_1',
    'DERS_2' : 'ders_2',
    'DERSEExample' : 'dersExample'
}
```

Note: configuration settings for an individual DER-S should be contained within the DER-S and not in the MCConfiguration class. This decouples the DER-S from the MC when it is removed from the simulation by removing the dictionary added above. For example, for a DER-S that reads data from a log file, the file path should be contained in

an attribute in the DER-S class, not MConfiguration; however, the file directory may be read from the MConfiguration class and used by the DER-S.

### **B.4.5 Example DER-S Design: RWHDEERS**

The following section describes the design process used to develop the Resistive Water Heater DER-S, starting from a high-level overview of processes and building to as a low-level description of the functions written to accomplish the specifications.

#### **B.4.5.1 Specifications**

RHWDEERS was developed specifically to provide an interface between the GSP and the Modeling Environment. More specifically, it was designed to tap into water heater emulators used by the GSP to test controls within its ECS system. These emulators were designed to test the GSP's ability to dispatch DERs in a realistic manner by providing realistic, controllable models of water heaters. They provide a profile of a water heater's functionality over time including water temperature, usage profiles, and modifications to operating conditions and setpoints based on DER dispatch. At any given time, the emulated water heater is either importing or exporting energy, corresponding to whether or not the heating element is determined to be on or off.

This heating element condition can be translated directly into a load profile. When it is on, the water heater is consuming power equivalent to the label plate rating of the element; when it is off, power is zero. Each emulated water heater is assigned a unique identifier; furthermore, since the emulators are designed to test a Grid Service Provider that dispatches resources locationally, each emulated DER contains a locational identifier as well. Note that the DERMS and MC topology must be in agreement for these locational identifiers to mean anything; in the case of the GSP, we are using the same Bus identifiers for the DERMS and the MC as part of the testing specifications, but topological processing may be required in other situations. Since the DER Inputs can be updated in real time and contain operating data (power), unique identifiers, and locational information, they contain all the information necessary to design a DER-S utilizing these emulators.

While the requisite data exists in the DER emulators, there remains the matter of communicating the data from the emulators to the DER-S. The DER-S must be programmed to receive data from a DER Input via whatever communication protocol necessary. In this case, the communication scheme between the emulators and DER-S was developed expressly for RHWDEERS, so we selected a scheme that would be extremely simple to implement, troubleshoot, and use.

Each water heater emulator generates a CSV file and places it in a folder to be read by the DER-S; the path to this folder is configured in the attributes of the RHWDEERS class. The file name for each CSV file is generated by the following scheme:

- 'DER#####\_Bus\$\$\$\$.csv'
  - 'DER#####': the ##### represents the unique identifier for the individual DER Input. For example, the unique identifier could be an DER LFDI or a counter starting with 'DER00000' and counting up.

- 'Bus\$\$\$': the \$\$\$ is the locational identifier. In our test cases, we used the IEEE 13 node test feeder, so one example of a locational identifier would be 'Bus632'.

The contents of each CSV file are a single two column row, no header, with two cells. The first cell contains 'P' for "power", dictating that power is meant to be controlled; 'V' would represent "voltage", but this functionality is not implemented in either the GSP or MC at this time. The second cell is a numeric value representing the value of power (in watts) that the water heater is consuming. The files are generated prior to the MC startup process and updated in real time by the emulators; RWHDEERS needs to read each of these files, determine a DER-EM to assign each input to at the proper location, reread the contents once per timestep to retrieve the updated power usage for each, and update the DER-EMs with the proper consumption at the proper time.

### B.4.5.2 Process Design Overview

Given the specifications above, bidirectional communication between the emulators and DER-S is not necessary. So, no connections need to be made during the startup process, and the DER-S can read the files immediately and directly. The files provide information required during the assignment process, as well as load profiles used throughout the simulation process.

- **During Startup:** Once during the startup process, RWHDEERS needs to scan the directory containing the DER Input CSV files. The necessary data for assignment are unique identifiers and locational identifiers, both of which are located by design in the file name. Therefore, RWHDEERS must read each file name, parse the identifiers, and place them in a list for assignment. The assignment process will then read them in, assign DER-EM mRIDs to each unique identifier, and maintain that association data within the DERIdentificationManager. Importantly, RWHDEERS maintains its own identification table associating file names with unique identifiers; this removes the need to parse file names every timestep.
- **Processing During Simulation:** The water heater emulators update the information within the CSV files at their own pace asynchronously to RWHDEERS. Once per timestep, RWHDEERS opens each file in its internal identification table (see above) and reads the power value contained in each file. These values are placed in dictionaries along with the unique identifier of the DER Input associated with each file; this list of identifier/value dictionaries is then stored for use by the output processes.
- **Output Functions:** Once per timestep, the EDMTimekeeper will call for updated input requests from all active DER-Ss. The input requests are generated during processing and placed in a list, so the only real output function that's required is one that returns said list. Since the output format is standardized (see B.3.2), this will be the case for all DER-Ss, including RWHDEERS.

### B.4.5.3 RWHDEERS Implementation

The following section is adapted from the class outline in Appendix A and demonstrates, in short, how the processes in B.3.5.2 are implemented. For more detail reference section A.2.1.

#### B.4.5.3.1 RWHDEERS Attributes

- **der\_em\_input\_request:** Contains the list of updates to be sent to the DER-EMs for this timestep.
- **input\_file\_path:** The folder in which the RWHDEERS input files are located.
- **input\_identification\_dict:** Contains data associating unique identifiers with file names and locational identifiers.

#### B.4.5.3.2 RWHDEERS Methods

- **initialize\_der\_s():** Mandatory startup function. Calls `self.parse_input_file_names_for_assignment()` and nothing else. This makes the code more readable without having to dig through the function.
- **assign\_DER\_S\_to\_DER\_EM():** Mandatory startup function. Takes the unique identifiers and associated locational identifiers from `self.input_identification_dict` and feeds them to the `DERAssignmentHandler` to assign them all to DER-EMs.
- **parse\_input\_file\_names\_for\_assignment():** Startup function. Reads all the input files from the folder (`self.input_file_path`) and parses their file names into dictionaries. Each dictionary's key is the unique identifier, and the value is a nested dictionary containing the filepath as well as the locational identifier for each.
- **get\_input\_request():** Mandatory output function. Calls `self.update_der_em_input_request()` and returns the `self.der_em_input_request` generated by that function. This is also encapsulated for readability purposes.
- **update\_der\_em\_input\_request():** Input/processing function. First, it clears the `self.der_em_input_request` attribute to minimize redundant processing. Then, it iterates through the `self.der_em_input_request` attribute. For each unique identifier in the list, it opens the file in the associated filename value, reads in the power data, and stores it in a dictionary in the "input request" format used by all DER-Ss: the key is the unique identifier used by the `MCInputInterface` to look up the DER-EM mRID, and the value is the power value the DER-EM will be set to this timestep. Each input request dictionary is appended to the `self.der_em_input_request` attribute, which upon completion of this method, will contain the updated DER states that need to be sent to the DER-EMs during this time step.

## Appendix C Product Requirements

Product Requirement Keywords	Description
SHALL	Indicates the development of this specification for this prototype is expected, without exception.
SHOULD	Indicates the development of this specification for this prototype is expected, but may not be completed.
MAY	Indicates the development of this specification for this prototype is not expected, but should be considered.

### C.1 ME Product Requirements

PR	Description
ME01	The ME SHALL allow for testing of the effects of a DERMS with an electrical grid model.
ME02	The ME SHALL simulate large numbers of DER-EMs
ME03	The ME SHALL be extensible to many DER types.
ME04	The ME SHALL provide a simulation of a GO.
ME05	The ME SHALL generate data logs.
ME06	The ME MAY be extensible to a variety of DERMS.
ME07	The ME MAY be extensible to a variety of grid models.

### C.2 DER-S Product Requirements

PR	Description
DER01	The DER-S SHALL provide electrical data to the MC necessary to generate control inputs to DER-EMs.
DER02	The DER-S SHALL have a configurable/reprogrammable API between external DER representations and itself.
DER03	The DER-S SHALL provide unique identifiers for its respective DERs.
DER04	The DER-S SHALL provide locational/topography information for its respective DERs.
DER05	The DER-S SHOULD be able to receive physical DER states
DER06	The DER-S SHOULD be able to receive simulated or emulated DER states
DER07	The DER-S SHOULD be able to receive data-represented DER states
DER08	The DER-S MAY have the capability to receive direct control messages from a DERMS.

### ***C.3 GO Product Requirements***

<b>PR</b>	<b>Description</b>
<b>GO01</b>	The GO SHALL have awareness of EDM grid model states.
<b>GO02</b>	The GO SHALL determine an appropriate grid service based on system state data from the EDM.
<b>GO03</b>	The GO SHALL have an API between itself and the DERMS.
<b>GO04</b>	The GO SHALL be able to request grid services from the DERMS.
<b>GO05</b>	The GO SHOULD provide a method to permit the TE to request grid services.
<b>GO06</b>	The GO SHOULD provide feedback data to the DERMS
<b>GO07</b>	The GO MAY alert the MC if it loses communication with a DERMS.

### ***C.4 EDM Product Requirements***

<b>PR</b>	<b>Description</b>
<b>EDM01</b>	The EDM SHALL be implemented in GridAPPS-D.
<b>EDM02</b>	The EDM SHALL include a database of grid models to be used in simulations.
<b>EDM03</b>	The EDM SHALL calculate new grid states at regular intervals.
<b>EDM04</b>	The EDM SHALL provide a means for measuring electrical characteristics within the simulated grid model.
<b>EDM05</b>	The EDM SHALL have configurable start time and duration.
<b>EDM06</b>	The EDM SHALL include non-DER asset models.
<b>EDM07</b>	The EDM SHALL include DER-EMs that are generalizable to a variety of DER types (including loads, sources, and storage assets)
<b>EDM08</b>	The EDM SHALL have the capability to add DER-EMs to existing grid models.
<b>EDM09</b>	The EDM SHALL model an electrical distribution system including unbalanced components.
<b>EDM10</b>	The EDM SHALL provide a unique identifier to each DER-EM.
<b>EDM11</b>	The EDM SHALL track locational/topological data for each DER-EM for assignment purposes.

## ***C.5 MC Product Requirements***

<b>PR</b>	<b>Description</b>
<b>MC01</b>	The MC SHALL coordinate simulations and input/output communications.
<b>MC02</b>	The MC SHALL provide access to grid states data to the GO.
<b>MC03</b>	The MC SHALL provide input and output interfaces.
<b>MC04</b>	The MC SHALL recognize if DER-Ss have changed state since the prior timestep.
<b>MC05</b>	The MC SHALL update DER-EMs as necessary.
<b>MC06</b>	The MC SHALL retrieve grid states data from the EDM at regular intervals.
<b>MC07</b>	The MC SHALL use a defined time step size for coordination purposes.
<b>MC08</b>	The MC SHALL provide an automated method to assign DER-EMs to DER-Ss (based on locational data.)
<b>MC09</b>	The MC SHOULD produce and store timestamped logs of grid state and operational data.
<b>MC10</b>	The MC MAY have the ability to inform/alert the TE.
<b>MC11</b>	The MC MAY have a closeout process.