Smart Charging Algorithms for Electric Vehicles considering Voltage and Thermal Constraints

This is a short documentation for the presented Java SmartCharging files.

License

Used Open Source Software in This Product

This program uses the following open source software:

- SQLite JDBC Driver by Xerial. Apache Software License, Version 2.
  https://github.com/xerial/sqlite-jdbc
  (https://www.apache.org/licenses/LICENSE-2.0)
- FontAwesomeFX. Apache Software License, Version 2.
  https://bintray.com/jerady/maven/FontAwesomeFX/11.0.0
  (https://www.apache.org/licenses/LICENSE-2.0)
- JFoenix. Apache Software License, Version 2.
  https://github.com/jfoenixadmin/JFoenix
  (https://www.apache.org/licenses/LICENSE-2.0)
- EasyModbusTCP by Rossmann-engineering. GNU General Public License 3.
  https://sourceforge.net/projects/easymodbustcp-udp-java/
  (https://www.gnu.org/licenses/gpl-3.0.de.html)

Documentation

In order to perform a simulation or a real test with an electric vehicle (EV), settings can be adjusted in the **MainController** and in the **Controller** class.

MainController Class

In the **MainController** class, objects are initialized and methods are called. This **MainController** class is necessary due to having a Graphical User Interface (GUI). There are also two threads in this class: The **operationalConditions** thread and the **simulation** thread.

operationalConditions Thread

The **operationalConditions** thread is synchronised with the **simulation** thread and by means of the database, voltage value data and transformer loading data can be imported for the simulation.

simulation Thread

In the **simulation** thread, the smart charging algorithms are called via the method *controller.chooseAlgorithm.* The parameters necessary are the number of the algorithm ("ChooseAlgorithm"), specified in the property file, the current time unit and the current time slot. This structure has been developed so that in later versions of the program, the desired charging algorithm can be chosen easily via the GUI. In the current program version, there is no need to change anything in this line of code

```
controler.chooseAlgorithm(Integer.parseInt(prop.getProperty("ChooseAlgorithm")), time, timeSlot);
```

The length of a time slot can be defined by the modulus operator. In this case, 10 time units represent 1 time slot.
Important: If you want to change the sampling time or the length of a time slot, you must adjust each method and definition that are dependent on the time. For example, the formula for the amount of energy charged (loadAmount) needs to be adjusted if you want to run the program in real-time.

```
if (time % 10 == 0) {
        timeSlot++;
}
```

After each execution of the charging algorithms, relevant data is stored in the database and the cars are charged virtually. The methods regarding the database can be modified in the **SQLiteDB** class. To store data, a corresponding table in the database must be available. Either a table has been created before or a method must be called in the initialize method of the **MainController** class.

```
@Override
public void initialize(URL arg0, ResourceBundle arg1) {
db.createNewTableResidentialArea();
}
```

In order to access the database, the path to the database file must be set in the **SQLiteDB** class.

```
String url = "jdbc:sqlite:C:/Users/user/eclipse-workspace/SmartCharging/SmartCharging.db";
```

To retrieve data from the measuring devices the following code has been used.

```
//Optional: Read measuring data from device
try {
        kocosV1_car = clientKoCos2.ConvertRegistersToFloat(clientKoCos2.ReadInputRegisters(4402, 2));
        kocosV2_car = clientKoCos2.ConvertRegistersToFloat(clientKoCos2.ReadInputRegisters(4404, 2));
        kocosV3_car = clientKoCos2.ConvertRegistersToFloat(clientKoCos2.ReadInputRegisters(4406, 2));

        kocosI1_car = clientKoCos2.ConvertRegistersToFloat(clientKoCos2.ReadInputRegisters(5234, 2));
        kocosI2_car = clientKoCos2.ConvertRegistersToFloat(clientKoCos2.ReadInputRegisters(5236, 2));
        kocosI3_car = clientKoCos2.ConvertRegistersToFloat(clientKoCos2.ReadInputRegisters(5238, 2));

        kocosI1_load = clientKoCos1.ConvertRegistersToFloat(clientKoCos1.ReadInputRegisters(5234, 2));
        kocosI2_load = clientKoCos1.ConvertRegistersToFloat(clientKoCos1.ReadInputRegisters(5236, 2));
        kocosI3_load = clientKoCos1.ConvertRegistersToFloat(clientKoCos1.ReadInputRegisters(5238, 2));
} catch (IllegalArgumentException | ModbusException | IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
}
```

Via *Thread.sleep(xx);* the time between two iterations of the main loop can be specified. In case you want to perform a real-time simulation, sleep times of about 600 (ms) have been considered appropriate.

initialize Method in MainController

Upon initialising the **MainController**, the initialize method is called.

```
@Override
public void initialize(URL arg0, ResourceBundle arg1) {
...
}
```

There, relevant objectives for the simulation are initialized. At first, the Modbus connection to the electric vehicle charge controller (EVCC) and the measuring devices (KoCoS) are established.

```
//Modbus: Connect
try {
        //EVCC
        clientEVCC1.Connect("129.217.210.206", 502);
        //EVCC needs adress to be 180
        clientEVCC1.setUnitIdentifier((byte) 180);

        clientEVCC2.Connect("129.217.210.205", 502);
        //EVCC needs adress to be 180
        clientEVCC2.setUnitIdentifier((byte) 180);

        //KoCos
        clientKoCos1.Connect("129.217.210.194", 502);
        clientKoCos2.Connect("129.217.210.193", 502);
} catch (UnknownHostException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
} catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}

try {
System.out.println("inetadress get local host: " + InetAddress.getLocalHost());
} catch (UnknownHostException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}
```

Next, charging stations are initialized. You can choose to either instantiate a pure virtual charging station or a charging station that is connected to an EVCC. The parameters passed for the charging stations are the id, the connection parameter of the charging station (1 = 1-Phase a, 2 = 1-Phase b, 3 = 1-Phase c, 4 = 3-Phase) and optionally the corresponding Modbus client.

```
ChargingStation cStationVirtual = new ChargingStation(0, 4);              //Virtual
or
ChargingStation cStationReal = new ChargingStation(0, 4, clientEVCC1);    //Real
```

Then, electric vehicles are initialized. The parameters necessary are:

Name
Maximal charging current [A]
Maximal charging power [kW] (irrelevant for the program)
Capacity [Ah]
SOC [%]
Rated voltage of the battery [V]
Conncection: 1-Phase (false) or 3-Phase (true)

The following example initializes a 1-Phase Nissan Leaf.

```
ElectricVehicle EV1 = new ElectricVehicle ("Leaf", 16, 3700, 67, 25, 360, false);
```

To assign an EV to a charging station, a schedule object is necessary. This object stores the arrival and the departure time of an EV.

```
Schedule scheduleEV1 = new Schedule (117, 188); //Arrival 19:30; Departure (+1) 7:20
```

Then, the EV can be assigned to a charging station. The third parameter is the amount of energy desired of the EV.

```
cStationVoltThermal.get(0).assignEV(EV1, scheduleEV1, 25000);
```

A ChargingStation object type has the two boolean variables **controlTMBS** and **touControl**. These variables indicate, if the charging station will perform thermal management or basic scheduling (**controlTMBS**) and if the charging station participates in TOU-scheduling (**touControl**).

```
cStationVoltThermal.get(0).setControlTMBS(true);          //Charging station will perform TM or BS
cStationVoltThermal.get(0).setTouControl(false);          //Charging station does not participate in
                                                          TOU scheduling
```

Thresholds for voltage and thermal control can be adjusted in the property file or in the initialize method of **MainController**.

Controller Class

In the **Controller** class, the different smart charging algorithms are defined as methods. Currently, the program supports four charging algorithms, namely Voltage Management, Thermal Management, Basic Scheduling and TOU Scheduling. To perform smart charging with a charging station, settings must be set in the **MainController** and **Controller** class together.

As described above, after initializing a ChargingStation type object, the two variables **controlTMBS** and **touControl** must be defined in **MainController** for the corresponding charging station.

Complementary, in the method *adjustAlgorithms(int k)* of **Controller**, settings have to be set according to the desired specifications of the test run. Hereinafter, exemplary settings of both the boolean variables and of the *adjustAlgorithms(int k)* method for different simulation settings are presented.

Simulation with Uncontrolled Charging Stations:

```
MainController:
            cStationVoltThermal.get(0).setControlTMBS(false);
            cStationVoltThermal.get(0).setTouControl(false);

            …

Controller:
      public void scheduling (int k) {
            //Does not matter.
      }
```

Simulation with Charging Stations with Thermal Management:

```
MainController:
            cStationVoltThermal.get(0).setControlTMBS(true);
            cStationVoltThermal.get(0).setTouControl(false);

            …

Controller:
      public void scheduling (int k) {
      for (int l = 0; l < cStationVoltThermal.size(); l++) {
            if (cStationVoltThermal.get(l).getCurrentEV() != null && cStation-
            VoltThermal.get(l).getLoadAmount() < 25000 && cStationVoltTher-
            mal.get(l).getLadeplan().getStartZeitpunkt() <= k && cStation-
            VoltThermal.get(l).isControlTMBS() == true) {
                  cStationVoltThermal.get(l).setDecisionVectorAtI(k + 1, 2);
            }
      }

            //Define a threshold for basic scheduling out of limit
            thresholdScheduling = 100000000;
      }
```

Simulation with Charging Stations with Basic Scheduling:

```
MainController:
              cStationVoltThermal.get(0).setControlTMBS(true);
              cStationVoltThermal.get(0).setTouControl(false);

              …

Controller:
       public void scheduling (int k) {
              basicScheduling(k);
       }
```

Simulation with Charging Stations with TOU Scheduling:

```
MainController:
              cStationVoltThermal.get(0).setControlTMBS(false);
              cStationVoltThermal.get(0).setTouControl(true);

              …

Controller:
       public void scheduling (int k) {
              touScheduling(k);
       }
```

Simulation with Charging Stations with combined TM and TOU Scheduling:

```
MainController:
              cStationVoltThermal.get(0).setControlTMBS(true);
              cStationVoltThermal.get(0).setTouControl(true);

              …

Controller:
       public void scheduling (int k) {
              touScheduling(k);

              //Define a threshold for basic scheduling out of limit
              thresholdScheduling = 100000000;

       }
```

Naturally, different simulation settings can be combined. For example, if you want to do a test run with a share of uncontrolled charging stations and a share of charging stations controlled with TM.

Simulation with Uncontrolled Charging Stations and Controlled TM Charging Stations:

```
MainController Uncontrolled Charging Stations:
            cStationVoltThermal.get(0).setControlTMBS(false);
            cStationVoltThermal.get(0).setTouControl(false);

            …

MainController Controlled TM Charging Stations:
            cStationVoltThermal.get(1).setControlTMBS(true);
            cStationVoltThermal.get(1).setTouControl(false);

            …

Controller:
      public void scheduling (int k) {
      for (int l = 0; l < cStationVoltThermal.size(); l++) {
            if (cStationVoltThermal.get(l).getCurrentEV() != null && cStation-
            VoltThermal.get(l).getLoadAmount() < 25000 && cStationVoltTher-
            mal.get(l).getLadeplan().getStartZeitpunkt() <= k && cStation-
            VoltThermal.get(l).isControlTMBS() == true) {
                  cStationVoltThermal.get(l).setDecisionVectorAtI(k + 1, 2);
            }
      }

            //Define a threshold for basic scheduling out of limit
            thresholdScheduling = 100000000;
      }
```

In the database file you will find exemplary results for different simulation settings.

Define Amount of Energy

There are some methods in **Controller** that use the amount of energy (loadAmount). These methods must be adjusted when using different amount of energy desired than the default 25 kWh.

Exemplary lines of code that need adjustment:

```
if (cStationVoltThermal.get(l).getCurrentEV() != null &&
cStationVoltThermal.get(l).getLoadAmount() < 25000 &&
(cStationVoltThermal.get(l).getLadeplan().getStartZeitpunkt() <= k) &&
cStationVoltThermal.get(l).isControlTMBS() == true) {

if (cStationVoltThermal.get(j).getLoadAmount() >= 25000) {
      if (optScheduleQueque.contains(cStationVoltThermal.get(j))) {
            optScheduleQueque.remove(cStationVoltThermal.get(j));
      }

if (cStationVoltThermal.get(k).getCurrentEV() != null &&
cStationVoltThermal.get(k).getLoadAmount() < 25000 &&
cStationVoltThermal.get(k).getLadeplan().getStartZeitpunkt() <= i)

if (cStationVoltThermal.get(k).getCurrentEV() != null &&
cStationVoltThermal.get(k).getLoadAmount() < 25000 &&
cStationVoltThermal.get(k).getLadeplan().getStartZeitpunkt() <= i  &&
cStationVoltThermal.get(k).getDecisionVectorAtI(i) == 2)
```

Transmit Charging Current to EVCC

In order to transmit the virtual charging current to the electric vehicle charging controller and thus to the EV, the method *controlAlgorithm (int i)* of **Controller** must be adjusted. For all charging station connected to an EVCC, the following lines of code must be substituted with each other.

For a virtual charging station use this code:

```
cStationVoltThermal.get(k).setCharCurr(cStationVoltThermal.get(k).getMostRestrictiveCurrLimitVoltThermal());
```

For a real charging station use this code:

```
cStationVoltThermal.get(k).setCharCurrReal(cStationVoltThermal.get(k).getMostRestrictiveCurrLimitVoltThermal());
```