EVLib: A Library for the Management of the Electric Vehicles in the Smart Grid

Sotiris Karapostolakis,
Emmanouil S. Rigas,
Nick Bassiliades
{skarapos, erigas, nbassili}@csd.auth.gr
Department of Informatics,
Aristotle University of Thessaloniki,
54124, Thessaloniki, Greece

Sarvapali D. Ramchurn
sdr1@soton.ac.uk
Electronics and Computer Science,
University of Southampton,
Southampton, SO17 1BJ, UK

ABSTRACT

EVLib is a Java library for the management and simulation of a number of Electric Vehicle (EV) activities, at a charging station level, within a Smart Grid environment. EVLib aims to solve interoperability issues between a number of Artificial Intelligence (AI)-related techniques already applied in this field. Thus, it provides a simple, yet efficient interface for the management of all major EV-related activities such as the charging and dis-charging of batteries, as well as the battery swapping. Moreover, a large number of parameters, such as the number of chargers, the waiting queues, and the available energy can be easily configured. On top of this, the library supports the simultaneous operation of many EV activities through the efficient use of threads. Finally, the library's efficiency and scalability have been tested in realistic scenarios, while the correctness and safety of the code have been verified using state of the art tools.

CCS Concepts

•**Software and its engineering** → *Software design engineering;* •**Computing methodologies** → *Artificial intelligence;*

Keywords

Electric Vehicle, Java Library, Charging, Dis-charging, Battery Swap, Smart Grid

1. INTRODUCTION

Electric vehicles (EVs) are entering our daily lives fast. It is estimated that approximately 740 thousand EVs have already been deployed worldwide, while the target set by the International Energy Agency is 20 million fully electric vehicles to be deployed by 2020 [2]. However, in order to ensure that the large-scale deployment of EVs results in a significant reduction of CO_2 emissions, it is important that they are charged using energy from renewable sources (e.g., wind, solar). Crucially, given the intermittency of these sources, mechanisms as part of a Smart Grid [1], need to be developed to ensure the smooth integration of such sources in our energy systems. EVs could potentially help by storing energy when there is a surplus, and feed this energy back to the grid when there is demand for it [3].

Indeed, the ability of EVs to store energy while being used for transportation [4] represents an enormous potential to make energy systems more efficient. On the one hand, given that vehicles drive only for a small percentage of the day, and considering the fact that EVs are equipped with large batteries, they could be used as storage devices when parked (i.e., as part of Vehicle-to-Grid (V2G) schemes [3]), and thus dramatically increase the storage capacity of the network. On the other hand, given that large numbers of EVs need to charge on a daily basis, (40% of EV owners in California travel daily further than the range of their fully charged battery [5]) if EVs charge as and when needed, they may overload the network. For this reason, new scheduling mechanisms are required to be able to manage the charging of EVs –Grid-to-Vehicle (G2V)– while considering the constraints of the distribution networks within which EVs need to charge [8]. Similarly, battery swap schemes which can be used instead of simple battery charge and can minimize waiting times at the charging stations [7], must also be efficiently designed and smoothly integrated to the grid.

Advanced Artificial Intelligence (AI) techniques (e.g., to mathematical programming based optimization, electronic markets and coalition formation) have been proven to be efficient in solving a number of EV-related problems, and a large number of such solutions already exists [6]. However, one of the key findings of [6] is that interoperability between various technologies and techniques is missing, and it is vital for successful large scale EV deployments. Thus, there is a need EV technologies to be able to work seamlessly and efficiently together. Different types of chargers should be able to work with all EV models, and data exchanged between entities (EVs, charging points, network operators) should have an understandable by all format and meaning. The EVLib[1] library tries to solve such problems, as it provides a simple, yet efficient, interface for the management of the basic EV activities (i.e., charging, dis-charging, battery swap) in a charging station level, while making use of a plethora of (renewable) energy sources. Precisely, it seeks the simula-

[1]http://sourceforge.net/projects/evliblibrary

SETN '16, May 18-20, 2016, Thessaloniki, Greece

DOI: http://dx.doi.org/10.1145/2903220.2903225

tion of charging stations that are linked to a Smart Grid and use multiple energy sources. A long term aim is this library to be used for the development of a charging station's information system which will communicate with the energy provider's central information system to allow for optimal energy usage and maximal EV satisfaction. Finally, the library's efficiency and scalability have been tested in realistic scenarios, while the correctness and safety of the code have been verified using state of the art tools.

2. THE EVLIB LIBRARY

EVLib is implemented in the JAVA programming language, and its main goal is to manage the charging, discharging and battery swap functions related to EVs and support their integration into a single charging station. The library supports a large number of functions to properly manage EV-related activities. There are three main functions, as well as a number of secondary ones, while each function is executed in 2 phases, namely the pre-processing and the execution. Figure 1 provides the library's detailed class diagram. In all functions, a set of discrete time points is used, while the actual duration of each time point can be defined by the user. The main functions are as follows:
1) **Charging (G2V technology)**: There are 2 types of charging depending on the charging time, namely the fast and the slow charging. The execution of a charging event requires first the pre-processing phase where a quest for an empty charger and available energy is performed. If the pre-processing phase is successful, the execution phase begins.
2) **Battery swap**: The pre-processing phase requires for a battery with enough range to be available in the charging station. If such a battery is found, the execution function can be called and the battery is swapped into the EV.
3) **Discharge (V2G technology)**: Similarly to a charging event, a discharging event first demands the pre-processing phase where a quest for an empty dis-charger is made. If this phase is successful then the execution begins.

The library also supports a number of secondary functions: The creation of a charging station, as well as the creation and integration of a charger, dis-charger or battery swapper in the station. Additional operations are the recharging of batteries which are later to be swapped into EVs, as well as the ability to add new batteries to the storage in order seamless operation of the battery exchange process to be achieved. Finally, the total cost of the charging, discharging and battery swapping can be calculated based on a series of costs (e.g., energy cost) defined by the user.

On the creation of the charging station, 4 waiting lists are created. One for the charging events which want fast charging, one for the charging events which want slow charging, one for the discharging events, and one for the vehicles waiting for battery exchange. Each list can be managed either automatically by the library using the default settings, or manually by the user. Moreover, each time an event is added to a list, an expected maximum waiting time is calculated, and once an event is executed these times are updated.

Moreover, the library lets the user manage and use a plethora of energy sources, through the use of an energy warehouse, for the supply of energy to the charging station. The library includes 6 different relevant classes, where each class describes a different energy source. The amount of available energy in the warehouse is updated in a regular basis. The energy in the warehouse can either be updated

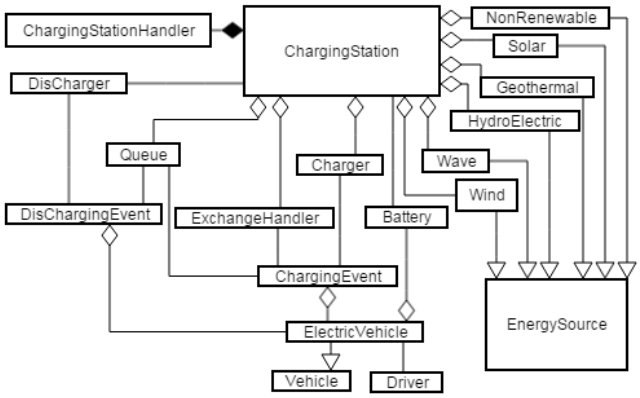

Figure 1: EVLib Class Diagramm

in predefined time points during the day, or the user can select these time points manually. The user can also rank the sources of energy used for a battery charge (e.g., solar energy may be preferred to wind).

2.1 Structure of EVLib

Here, the main classes of the library are presented.

2.1.1 ChargingStation

ChargingStation class describes a charging station and is a central class in EVLib. It includes a plethora of fields, where some of the most significant ones are the following: 1) *chargingRatioSlow* field for the rate of slow charging, 2) *unitPrice* field for the price of each energy unit for charging, 3) *automaticQueueHandling* field for the option of how each list is going to be handled. Most of the fields have relevant methods to define their value and their return.

The class provides 3 constructors: The first, ChargingStation(int id, String name, String[] kinds, String[] source, float[][] energAm) demands an id, a name, a table with the kinds of charging that supports, a table with the sources of energy that are used, and a table with the currently available amount of energy from each source in the energy storage. The second, ChargingStation(int id, String name, String[] kinds, String[] source) is similar to the first one, but does not need the table with the given energy sources. The third, ChargingStation(int id, String name) only requires an id and a name for the charging station.

ChargingStation class has a number of methods, where the most important ones are the following: 1) updateDisChargingQueue(DisChargingEvent d) inserts a discharging event to the waiting list, 2) checkChargers(String k) checks for a kind of charger according to the argument, 3) insertEnergySource(EnergySource z) inserts an energy source to the charging station, 4) customEnergySorting(String[] energies) sorts the types of energy sources in the row they are defined in the argument, 5) setSpecificAmount(String v, float q) sets an amount of energy in a specific energy source, finally 6) batteriesCharging(String kind) charges the batteries which are to be used for exchange with the type of charging given in the argument

2.1.2 Charger

An object of this class represents a charger of a charging station. The most important fields are the following: 1) *kindofcharging* field for the kind of charging the charger supports, 2) *busy* field shows if a charger is busy or not, 3)

commitTime field for the time the charger is occupied.

Charger class provides one constructor, Charger(int id, ChargingStation station, String kindofcharging), as well as methods: 1) executeChargingEvent(ChargingEvent ch) executes a charging event and, 2) handleQueueEvents() executes the first charging event in the waiting list. Note that, the DisCharger and ExchangeHandler classes operate similarly to the Charger class.

2.1.3 ChargingEvent

ChargingEvent class is responsible for the representation of a charging event. The fields that are used are the following: 1) *amEnergy* field which shows the amount of energy the event needs, 2) *chargingTime* field is the duration of the charging event, 3) *mode* field which can take 5 values each of them indicating the state of a charging event (i.e., ready to be executed, inserted to the waiting list, cannot be executed, just created, executed).

ChargingEvent class disposes 3 constructors. The first constructor ChargingEvent(ChargingStation station, ElectricVehicle vehicle, float amEnerg, String kindOfCharging) demands the charging station the vehicle visited, the vehicle that is going to be charged, the energy that is asked for and the type of charging the user wants. The second constructor ChargingEvent(ChargingStation station, ElectricVehicle vehicle, String kindOfCharging, float money) is similar to the first one but instead of energy there is an argument that shows the money for which the user demands energy. The third constructor ChargingEvent(ChargingStation station, ElectricVehicle vehicle, String kindOfCharging) is for the battery exchange function. In the kindofCharging argument the user gives the "exchange" value. The main methods of this class are: 1) preProcessing() method which materializes the preliminary process before the charging, 2) execution() is competent for performing the charging.

2.1.4 ElectricVehicle

This class describes an electric vehicle and inherits from the Vehicle abstract class. It contains 2 fields which are: 1) *driver* field for the driver of the car and, 2) *battery* for the battery that is linked to the vehicle.

The class has 1 constructor, ElectricVehicle(int id, String brand, float cubism). The most significant methods are: 1) vehicleJoinBattery(Battery k) method links a battery with the vehicle and, 2) setDriver(Driver p) sets a driver to the electric vehicle.

2.1.5 Battery

This class is responsible for the representation of a battery and its main fields are: 1) *remAmount* field for the remaining amount of energy and, 2) *batteryCapacity* field for the capacity of the battery.

Battery class provides 2 constructors: The first, Battery(int id, float remAmount, float batteryCapacity) demands an id, a value for the remaining amount of energy in the battery, and a value for the capacity of the battery. The second, Battery(int id) requires only an id. Moreover, the most important methods are: 1) setBatteryCapacity(float u) sets a capacity for the battery, 2) setRemAmount(float r) sets a remaining amount for the battery, 3) reRemAmount() returns the remaining amount of battery's energy and, 4) reBatteryCapacity() returns the capacity of the battery.

2.1.6 EnergySource

The library disposes a sum of renewable and non-renewable energy sources that can provide energy to the charging station. Specifically there are 5 sub-classes, each of them representing a renewable source and 1 sub-class for the energy from non-renewable sources. All classes have the same interface and inherit this abstract class. It includes 1) an *id* field, 2) *station* field that stores the charging station the source is attached to. Each of the 6 classes that inherit EnergySource class contain an *energyAmount* field, which stores the energy that is going to be inserted after each update.

Every class (e.g., Solar) supports 2 constructors: The first one Solar(int id, ChargingStation station, float[] energyAmoun) demands an id, the charging station the source will be attached to, and a table with the amount of energy for some updates. The second, Solar(int id, ChargingStation station) requires an id and the charging station the source will be embodied. Moreover, EnergySource class has 2 methods: 1) reAmount(int num) returns the energy to be given in the charging station at a specific update and, 2) modifySpecificAmount(int num, float am) modifies a amount of energy that is going to be given at a specific update.

3. USING EVLIB

This section presents a step-by-step explanation of how to set up a charging station and carry out some functions with EVLib. In so doing, class EVLibEx1.java is created.

Step 1: Determination of the kind and the number of chargers. We create table kind and we use the argument "slow" to create a slow charger and the argument "fast" to create a fast charger in the chargers' table. **Step 2**: Definition of the energy sources that will be connected to the station. We create table source. Regarding the types of energy, the argument "wave" corresponds to a Wave (energy) object, "geothermal" to a Geothermal object and "nonrenewable" to a NonRenewable object. **Step 3**: We construct a table whose first dimension represents the number of energy sources and the second one the number of updates that we want initially to define the energy to be given to the station. In this case, we want only at the first update to provide energy, thus the second dimension will be equal to 1.

```
String[] kinds={"fast","slow","slow"};
String[] source={"wave","nonrenewable","wind"};
float energyAmounts[][] = new float[4][1];
for(int i=0;i<4;i++)
    energyAmounts[i][0] = 40;
```

Step 4: Creation of a charging station. We specify the number of (dis-)chargers and the number of simultaneous exchanges of battery that can be supported in the station.

```
ChargingStation station =
new ChargingStation(2,"Florida",kinds,
source,energyAmounts);
station.setNumOfSlots(2);
station.setExchangeSlots(2);
```

Step 5: A new vehicle, its battery and a new driver are defined. **Step 6**: The battery and the driver are linked to the vehicle.

```
ElectricVehicle vehicle =
new ElectricVehicle(4,"Honda",2000);
Battery b = new Battery(4,50,100);
```

```
Driver driver = new Driver(1,"Nick");
vehicle.vehicleJoinBattery(b);
vehicle.setDriver(driver);
```

Until now we dealt with forming the framework around the station. For running a charging event we can use the following code: **Step 7**: Creation of 1 charging event.

```
ChargingEvent event1 =
new ChargingEvent(station,vehicle,20,"fast");
```

Step 8: Pre-processing and execution phase of the charging event.

```
event1.preProcessing();
event1.execution();
```

Step 9: Construction of a discharging event.

```
DisChargingEvent event2 =
new DisChargingEvent(station,vehicle,25);
```

Step 10: Pre-processing and execution phase of the discharging event.

```
event2.preProcessing();
event2.execution();
```

Step 11: Construction of an exchange battery event.

```
ChargingEvent event3 =
new ChargingEvent(station,vehicle,"exchange");
```

Step 12: Pre-processing and execution phase of the battery exchange event.

```
event3.preProcessing();
event3.execution();
```

4. SOFTWARE TESTING

The library has been proven to achieve low execution times and good scalability (see Figure 2 for CPU usage times). Moreover, the library was checked for errors regarding the code development, errors about the view of the code or security holes that may exist using state of the art tools. In so doing, FindBugs[2] and CheckStyle[3] were utilized. FinbBugs is a tool which carries out static testing on Java code. Alongside, CheckStyle reports all cases which do not follow patterns of proper programming, related to the view and logical development of code. In addition to these programs, a number of unit tests, related to the basic classes(ChargingEvent, Charger, DisChargingEvent, DisCharger) were developed using JUnit 4. All the errors that were discovered were corrected.

5. CONCLUSIONS AND FUTURE WORK

As mentioned earlier in the text, Electric Vehicles is a continuous growing sector that is strongly related to the Smart Grids and the extended use of renewable energy sources. The bidirectional communication between EVs and the Smart Grid creates great opportunities for the IT and the power systems sectors, however in order advances to be achieved interoperability issues need to be solved. The EVLib tries to solve such problems by providing a simple, yet efficient interface for the management of all major EV-related activities such as the charging and dis-charging of batteries, as well as the battery swapping, while using energy from renewable sources.

[2]http://findbugs.sourceforge.net/
[3]http://checkstyle.sourceforge.net/

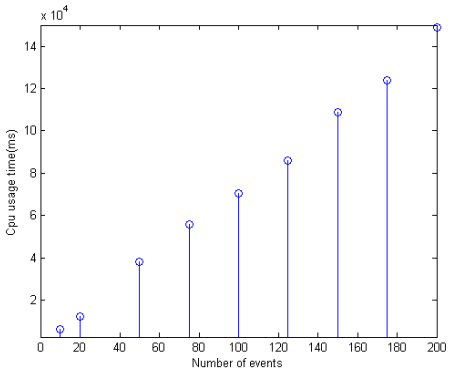

Figure 2: CPU usage time

Future work will focus on the integration of more advanced AI techniques, such as optimization, electronic markets, agent-based negotiation and coalition formation to further expand the library's capabilities. Moreover, mechanisms and techniques for managing uncertainty and increase fault tolerance will also be studied. Finally, a real world validation of the library where the already promising empirical results would be verified, would be proved useful.

Acknowledgment

Emmanouil Rigas gratefully acknowledges financial support from the Hellenic Artificial Intelligence Society (EETN) for attending this conference.

6. REFERENCES

[1] H. Farhangi. The path of the smart grid. *Power and Energy Magazine, IEEE*, 8(1):18–28, January 2010.

[2] IEA. Global ev outlook. Technical report, 2013.

[3] W. Kempton and J. Tomic. Vehicle-to-grid power fundamentals: Calculating capacity and net revenue. *Journal of Power Sources*, 144(1):268 – 279, 2005.

[4] W. J. Mitchel, C. E. Borroni-Bird, and L. D. Burns. *Reinventing the automobile: Personal urban mobility for the 21st century*. MIT Press, 2010.

[5] M. A. Nicholas, G. Tal, and J. Woodjack. California statewide charging survey: What do drivers want? *92nd Annual Meeting of the Transportation Research Board*, 2013.

[6] E. Rigas, S. Ramchurn, and N. Bassiliades. Managing electric vehicles in the smart grid using artificial intelligence: A survey. *Intelligent Transportation Systems, IEEE Transactions on*, 16(4):1619–1635, Aug 2015.

[7] E. S. Rigas, S. D. Ramchurn, and N. Bassiliades. Algorithms for electric vehicle scheduling in mobility-on-demand schemes. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 1339–1344, Sept 2015.

[8] E. S. Rigas, S. D. Ramchurn, N. Bassiliades, and G. Koutitas. Congestion management for urban ev charging systems. In *Smart Grid Communications (SmartGridComm), 2013 IEEE International Conference on*, pages 121–126, 2013.