Model Implementation for the Extendable Open Source Power System Simulator RAPSim

Manfred Pöchacker, Wilfried Elmenreich
Institute of Networked and Embedded Systems / Lakeside Labs
Alpen-Adria Universität Klagenfurt, Austria
Email: manfred.poechacker@aau.at, wilfried.elmenreich@aau.at

***Abstract*—The open power system simulation framework RAPSim provides interfaces for extending and adapting simulation models and algorithms. This is especially useful for researchers working with experimental custom models. The code of RAPSim is available under an open source license which allows users to publish extensions together with the original code. This paper gives an overview on the software architecture of RAPSim and introduces the necessary steps to implement a new customized model by the example of a wind turbine model, which can be implemented with three short code snippets and by specifying an icon for the graphical user interface. As a result, the new wind turbine model is available for simulation models and can be handled with full GUI support.**

Keywords Microgrids, Open Source Simulation, Power System Simulation, Renewable Energy Sources

I. Introduction

Integration of renewable and green power sources is important for power grids all over the world. The prospect of reduced carbon emission, cost-efficient and distributed power production are some of the enhancers of this development. As the advantages of renewable power generation are appreciated, sustainable installation of power systems requires specific design fitting the local conditions. A broad field of environmental conditions must be considered in the planning phase of a micro power system. This includes for instance available resources, climate and weather conditions, expected power demand profile, user habits, or fault and maintenance scenarios. Both, research and engineering requires smart grid simulations that match these diverse specifications. In particular, extensive simulations are needed to demonstrate the abilities of a highly reconfigurable energy grid [1].

Simulation of microgrid scenarios is a field that requires combination of experts and methods from various fields. Physical continuous models, discrete communication models, game theory, and statistical modeling should complement each other [2]. Current research works on combination of different approaches or integrates methods from other fields into established tools. A comparison for open power system simulation software is given in [3]. Besides high end research tools and high performance commercial software there is an application field of easy to use and flexible simulation software. This basically aims at interested users that cannot spend much money on licenses and do not have with sophisticated programming skills. As for instance in teaching, the goal would be to facilitate using such a tool in a class. RAPSim is a microgrid simulation tool designed with such features in mind. The combination of renewable power source simulation and basic power system features makes it interesting especially for microgrid simulation. A general description of the software is presented in [4]. RAPSim is open software and the source code is publicly available at *http://rapsim.sourceforge.net/*.

All the features in RAPSim are accessible via a clearly laid-out GUI which is a distinguishing feature [3] compared to many other power system simulators like MatPower [5], UWPFLOW [6], IPSYS [7], MatDyn [8], and GridLAB-D[1][10]. As many publications demonstrate the field is rapidly progressing [11], [12], [3], [13], [14], [15]. RAPSim focuses on microgrid simulation (island and grid connected [16]) with different types of (renewable) generation and residential loads. It is intended for usage in classroom without accruing of license costs and for suitable research.

While using RAPSim via its graphical user interface aims at users without programming expertise, it is possible to extend RAPSim with additional models, for example for distributed energy resources or consumer patterns by writing Java classes that fit into the structure of the project. RAPSim's architecture provides flexibility for changing parts of any simulation model; adapting the objects and their models to the local conditions is possible while the grid wide control algorithm remains unaffected. It allows the inclusion of measured data, e. g., for local weather conditions or user load curves from real measurement campaigns such as [17]. Many functions and features are created to simplify this and make it easy also for users with minimum programming experience. This paper gives an introduction for implementing such user specific models in RAPSim by giving two examples. The first example is based on a mathematical model, starting with the characteristic equation and following all the steps until detailed code explanations. The second example utilizes external data of residential demand to get a model within RAPSim.

The next section contains a brief description of the RAPSim

This work was performed in the research cluster Lakeside Labs funded by the European Regional Development Fund, the Carinthian Economic Promotion Fund (KWF), and the state of Austria under grants 20214/22935/34445 (Smart Microgrid Lab) and 20214/23743/35470 (Project MONERGY).

[1]There exist custom GUIs for GridLAB-D such as GridSpice [9]. By default, GridLAB-D is distributed without a user interface.

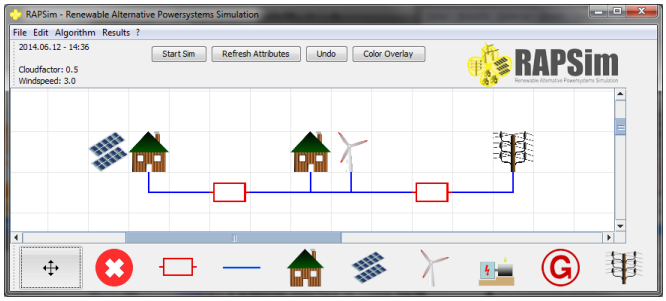

Fig. 1: A simple scenario within RAPSim of two grid connected households with wind and PV generation

project and its current capabilities. This is followed by a description of the relevant software components for model implementation in section III. The concept of the models and the two examples are content of Section IV. The first exemplary model is a wind turbine generator, the second one describes a load model of an average residential house. Finally, we give a conclusion and an outlook on future work.

II. The RAPSim Simulation Framework

The RAPSim simulation software provides several *grid objects* that can be placed in a simulation lattice. Grid objects can be any devices that produce or consume electrical power or are involved in power transportation and transformation. All the objects together form a *scenario* which is the topological representation. RAPSim adds time-based simulation to the scenario. Timing (starting and end time of simulation, increment) can be specified up to one minute resolution and span multiple days. The simulation framework is structured in two levels, a grid-wide one and one level of individual grid objects. The calculations at the grid wide level are called *algorithms*, e. g., power flow analysis. A single grid object is based on an *object model* for internal calculations. The simulation framework provides the following features:

a) A graphical interface to create the intended scenarios and to control the simulators functions.
b) Functions to save and load simulation scenarios in a generic XML format.
c) A time thread that models time of day and day of the year up to minute resolution.
d) Generation of output files in CSV-format. All object parameters can be selected in any combination to be written into a CSV output file for all time steps.
e) Weather simulation which can be done via stochastic models or emulated by measured data.
f) Topological grid analysis that identifies the objects within a bus and aggregates their parameter values.
g) Administration of algorithms for any grid wide calculations.
h) Administration of application-specific models that can be implemented and added by the user.

The interaction of these functions within a time step of the simulation is shown in Figure 2. The weather data in combination with the current time are given values or environment

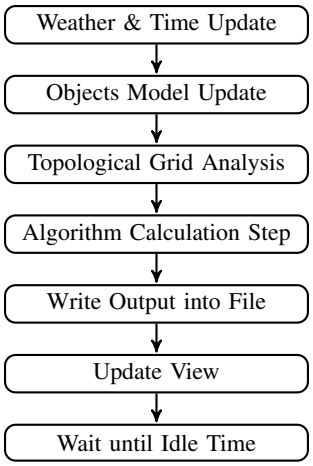

Fig. 2: Tasks executed within a simulation step of the time thread

for the following tasks. Many object models, like those for solar power plants, refer to this data. The topological grid analysis is only done in case the grid was modified since the last simulation step. Any grid modification during running simulations releases an interrupt on the time thread to redo the topological analysis. The user has the freedom to add, remove, modify, or relocate any grid object, even while the simulation is running. After the algorithm's function was performed the results are shown by updating the view and optionally, by writing to the output file. For fine-tuning the user's view of the simulation in the GUI, it is possible to set an *idle time* until the next simulation step allowing for an accelerated real-time view of a simulation.

The whole project is implemented in Java, therefore it is platform-independent. The grid objects can be specified for designated cases, and output files can be generated as required. Users can import the project into their preferred integrated development environment (IDE), for example eclipse[2]. By inheriting from abstract models, RAPSim can be quickly extended with new models which will be automatically included in the selection menus of the GUI. As we will show, extending RAPSim is straightforward: The implementation of new models requires some Java programming skills and some basic understanding of the software structure. An abstract Java class needs to be extended, the specific methods must be implemented and the classes must be added to the fitting code package The following paragraphs provide all necessary information to enable the reader for implementation of a Grid Object Model in RAPSim.

Figure 3 shows the property window of the grid object wind turbine power plant. The window name contains also the position in the simulation lattice and the bus which this object is assigned to. The upper part shows three object variables. The algorithm (which is simple power distribution in this case) sets the required object parameters visible. The part below is designated to the model of the object. It contains

[2]eclipse.org

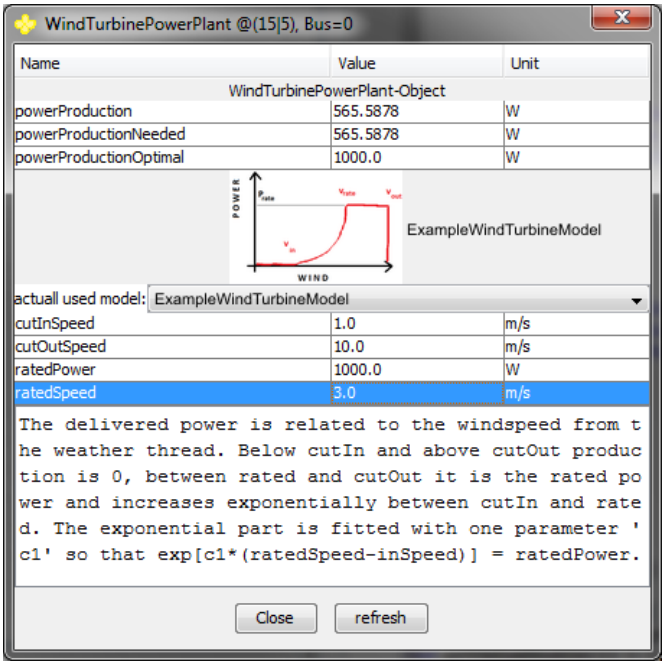

Fig. 3: Property window of a wind turbine power plant

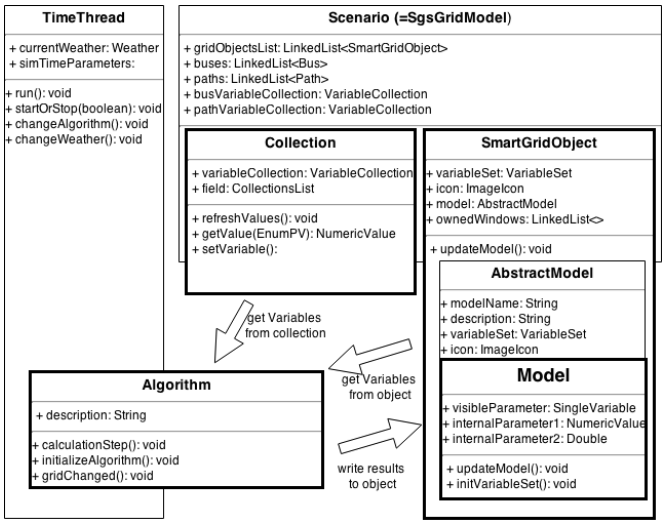

Fig. 4: Within the RAPSim software structure the grid object model is encapsulated in the object

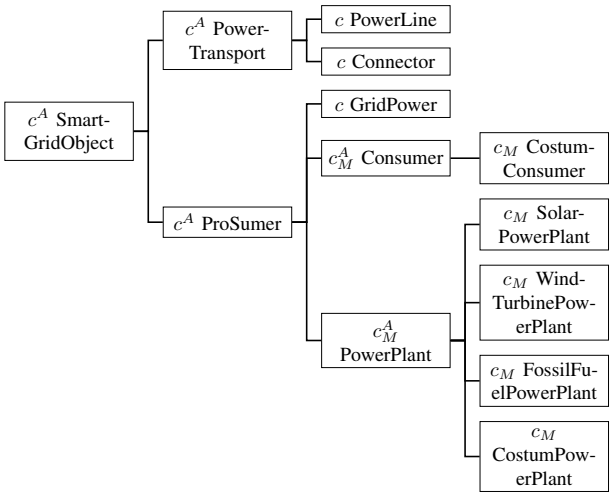

Fig. 5: The current inheritance tree of the grid objects in RAPSim. All abstract classes are labeled as c^A and classes that instantiate a model as c_M.

a selection menu for the different implemented models, an icon, the parameters of the model and a description field.

III. Software Structure of RAPSim

The general software structure is following the model-view-controller pattern. All the classes required for implementation of a new grid object model can be found in the packages `sgs.model.gridObjects` and `sgs.model.objectModels`. Figure 4 gives an overview about the main software parts involved in the simulation. The time thread initializes the algorithms and performs the calculation step as shown in Figure 2. The SgsGridModel, which represents the scenario, contains the smart grid objects as well as the collections. Smart grid objects that can handle models have an abstract model class associated. Any new model needs to inherit from this abstract model class.

A. Smart Grid Objects and their Object Models

The Smart Grid Objects is the general abstract class for all grid objects used in the simulation. The abstract object classes implement general functions for

- administration of the objects *parameter set* and its behavioral model,
- handling of the property window,
- generation of the object menu in the GUI, and
- displaying the object icon in the simulation panel.

The specific calculations are implemented in the object or in its protected attribute model. Thus, a model is encapsulated in its object. This enables the use of different models for the same object, like for instance a wind turbine which can be modeled by the power curve $P(v)$ of a specific type or by its geometry and the kinetic energy in the wind. The model calculates internal behavior while the object communicates its results at grid level via a standardized interface. The current software structure provides several grid object classes which inherit from the abstract class smart grid object as shown in Figure 5. This structure will be enriched by additional grid objects in future versions of the software project. The type of object implicitly declares some of the used parameters, like e. g., every power plant object has the parameter power production.

Figure 6 shows the currently implemented model structure which follows closely that of the objects from Figure 5. For all object classes c_M exists an abstract model class which needs to be extended to associate the implemented model with the specific object type. Abstract models organize the interaction with the object and provide general and GUI-related functions

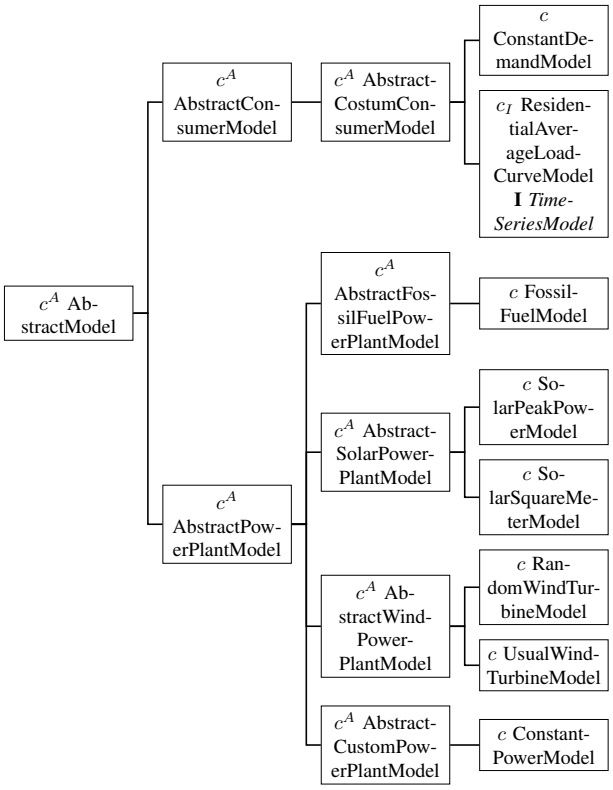

Fig. 6: Inheritance Tree of the currently implemented model classes

for their subclasses. The interface for time series models generated from external data is used in the residential average load curve model.

B. A Parameter Set of Single Variables with Numeric Values

To make a variable be integrated as a parameter into the GUI-functions the variable needs to be of type *single variable* and included in a parameter set. The objects as well as the models have an attribute of type *parameter set* which is a list of single variables. Only those parameters can be visible and optionally editable via the property window of the object, as shown in Figure 3. To add a single variable to the parameter set it needs to be specified by its name, by its (initial) value, its physical unit, and two options, which are boolean values for defining the variable to be visible and editable in the property window. The name of a model parameter can be any string but should be chosen in a meaningful way and fit the terms in the model description. The value must be of the type *numeric value*. Numeric value is a class created within RAPSim to deal with real as well as complex numbers. It provides functions for calculations as well as for conversion to the usual numeric Java data types. For common numerical operations, like addition or division, it is necessary to call the specific methods of the object instead of the usual numerical operators, since operators $+$ or $/$ do not work for this data type. The physical unit needs to be selected from the EnumUnit list which can be extended if required.

C. Algorithms and Collections

Algorithms do all operations for grid-wide calculation, for example a power flow analysis. The buses and paths of the power flow are lists of object and do have a *collection* of parameters within RAPSim. A collection is generally an aggregation of several parameters of grid objects. As an example, a grid topology analysis generates lists of buses and paths. Each bus aggregates several objects. The collection builds then an aggregated parameter set that holds the aggregated power of all the objects. Collections work, according to their name, by aggregating values from objects and can never write to objects. That means the algorithm can use collections as input but must return results to the objects directly. In RAPSim the user can select from the implemented algorithms via the GUI. Newly implemented algorithms are automatically integrated into the menu including their description. The implementation of an algorithm is more complex than writing a new model due to their multiple dependencies on various data from grid object models.

IV. Model implementation in RAPSim

By showing two models in detail we demonstrate implementation of object models within RAPSim. From the scenario in Figure 1 we present a model for a wind turbine and for a residential consumer. The first model is based on a mathematical model, in particular an equation with four parameters. The second model is using time series data from an external data source which must be adapted to the pace of the simulation time. There are three main steps in implementing a model.

1) Selection of the appropriate abstract model class to be extended and declare the name, description and icon for the model.
2) Initialization of the variables including the options to be visible and editable in the property window.
3) Definition of the update procedure for the variables which is the core of the model.

For each of these three tasks there is a designated place to implemented the code fragments. They are particularly explained for the wind turbine example in the following.

A. Wind turbine model

This subsection exercises the implementation of a model for wind turbines as shown in Figure 3. The output power of any wind turbine is a function of the wind speed v which is a parameter provided by the weather thread within RAPSim. The icon in the center of Figure 3 shows a general form of the power curve of wind turbines. The power production characteristic $P(v)$ of this wind turbine model is described by the equation

$$P(v) = \begin{cases} 0, & v < v_{cutin}, v > v_{cutout} \\ P_{rated}, & v_{rated} < v < v_{cutout} \\ e^{c_1(v-v_{cutin})}, & v_{cutin} < v < v_{rated}. \end{cases} \quad (1)$$

Where, v_{cutin} is the speed that the turbine starts working at, v_{cutout} , the wind speed at which the turbine stops working to prevent damages. Between v_{rated} and v_{cutout} the wind turbine

generates the rated power. The curve between v_{cutin} and v_{rated} can be generally fitted by any function. In Equation 1 this is done by one parameter so that $P(v_{rated}) = P_{rated}$ and $P(v_{cutin}) = 1W$. Parameters depend on the individual wind turbine type. The data can be obtained from the producer or must be measured directly. For simplicity, we implement a single parameter version. The following code listings implement Equation 1 in Java and add all required functions for model administration in RAPSim.

```
this.ratedPower = new SingleVariable(
    EnumPV.powerProductionOptimal,
    this.powerPlant.getPeakPower());
this.ratedPower.properties.set(true, false);
this.variableSet.add(ratedPower);
```

Listing 1: Definition of the rated power variable in the abstract power plant model. This single variable, which requires to be updated with the objects variable of the same name, must be named according the EnumPV list. The default options for variables defined in abstract classes are visible true and editable false.

The specification of the variable rated power and current power is done in the abstract power plant model, which manages the update with their equally named counterparts in the object. The (initial) value is loaded from the power plant object with the getPeakPower method. This keeps the value preserved when a different model gets selected.

```
12
13 //  is initialized in super class
14 //  private SingleVariable ratedPower;
15 //  private SingleVariable powerProduction;
16  private SingleVariable cutInSpeed;
17   private SingleVariable ratedSpeed;
18   private SingleVariable cutOutSpeed;
19   private double c1;
20
21  public UsualWindTurbineModel(
        WindTurbinePowerPlant powerPlant) {
22   super(powerPlant);
23   modelName = "UsualWindTurbineModel";
24   description = "The delivered power is related
        to the windspeed from the weather thread.
        Below cutIn and above cutOut production is
        0, " +
25     "between rated and cutOut it is the rated
          power and increases exponentially between
          cutIn and rated. " +
26     "The exponential part is fitted with one
          parameter 'c1' so that
          exp[c1*(ratedSpeed-inSpeed)] =
          ratedPower.";
27   icon = new
        ImageIcon(ClassLoader.getSystemResource(
        "Data2/WindTurbineUsualModel_ICON.png"));
28 }
```

Listing 2: The class definition contains the variable declaration, the constructor and defines other attributes.

Beside the rated power P_{rated} the model requires three wind speed parameters, v_{cutin}, v_{rated}, and v_{cutout}. Those are of type single variable to be accessible via the parameter set in the GUI. They are initialized in the initVariableSet method. The fitted parameter can be of type double. The model constructor requires the associated power plant object as argument and calls the super class constructor. Then the three arguments name, description and icon are defined for model administration. Evidently, name and description should be meaningful. The icon is recommended to be of type .png and 120 times 80 pixels wide.

```
31 @Override
32 protected void initVariableSet() {  // called in
       constructor of super class
33  this.ratedPower.properties.set(true, true);
34  this.powerProduction.properties.set(false,
       false);
35
36  this.cutInSpeed = this.initVariable("cut-in
       speed", new NumericValue(1.0),
       EnumUnit.meterPerSecond, true, true);
37  this.ratedSpeed = this.initVariable("rated
       speed", new NumericValue(3.0),
       EnumUnit.meterPerSecond, true, true);
38  this.cutOutSpeed = this.initVariable("cut-out
       speed", new NumericValue(10.0),
       EnumUnit.meterPerSecond, true, true);
39 }
```

Listing 3: The method initVariableSet() is called in the constructor of the super class.

The method initVariableSet is called in the constructor of the super class. It contains modification of the options for the parameters predefined in the super-classes, and initialization of the variables.

```
41 @Override
42 public void updateVariables(GregorianCalendar
       currentTime, Weather weather, int
       resolution) {
43
44  double windSpeed = weather.getWindSpeed();
45
46  if (windSpeed<cutInSpeed.getValueDouble() ||
       windSpeed>cutOutSpeed.getValueDouble()){
47   this.setPowerProduction(new NumericValue(0.0));
48  } else if (
       windSpeed>ratedSpeed.getValueDouble() &&
       windSpeed<cutOutSpeed.getValueDouble() ){
49   this.setPowerProduction(this.getRatedPower());
50  } else if
       (windSpeed>cutInSpeed.getValueDouble() &&
       windSpeed<ratedPower.getValueDouble()){
51   c1 = Math.log(ratedPower.getValueDouble()) /
        (ratedSpeed.getValueDouble()
        -cutInSpeed.getValueDouble());
52   NumericValue pValue = new
        NumericValue(Math.exp (c1*(windSpeed-
        cutInSpeed.getValueDouble()) ) );
53   this.setPowerProduction( pValue );
54  }
55 }
```

Listing 4: The method updateVariables contains the main calculations for the model.

The method *update variable* is executed at each time step within the model update. It needs the arguments about time and weather conditions. The calculation of the c_1 constant is redone each time step to capture parameter modifications via the GUI. Values of the predefined parameters are rounded to four digits and then forwarded to the associated object.

B. Model for Residential Average Loads

Residential loads in the distribution grid are often modeled statistically as averaged load profiles. Such models can be easily evaluated with measured data from a distribution feeder and are commonly used for evaluations of other models, e. g., in load shifting studies like [18], [19]. The average German load profiles H_0 as used in [18] are accessible within RAPSim through a consumer model. The German Federal Association of Energy and Water Industries (BDEW) provides this profile, called H_0 in a 15-minute resolution for the average electricity consumption of a norm German household. The dataset includes different profiles for working days, Saturdays and for Sundays (also valid for public holidays) as well as for the different seasons Winter, Summer and transition (for Spring and Autumn). The annual energy consumption of this average house is scaled to 1000kWh. The residential average load curve model makes these data usable within RAPSim. The necessary functions are:

- Reading the data file into an array.
- Selecting the respective time line for the currently simulated season and day of the week.
- Getting the corresponding value according to current simulation time.
- Aggregating or splitting values according to the simulation step size.
- Scaling the value according to the annual energy usage of the house, which is the only editable variable of the model.

There are many Java libraries which provide functions for the import of CSV files including time series. The Java.io package is used in the residential average load curve model.

V. Conclusion and Future Work

This paper described the main structure of RAPSim and gave instructions on how to extend the simulation by implementing user-specific models of renewable or distributed energy sources and consumers. The main feature of RAPSim is the simulation and aggregation of time depending power profiles (of consumption and generation) down to minutes resolution. The effort for the user on model implementation is kept small by a systematic use by abstract classes that handle all (internal) functions and by providing a general weather and climate simulation thread. The software design clearly separates the modeling of the grid objects from the power grid simulation. The simultaneous use and simple exchange of different models for the same type of object is easily possible. RAPSim supports the implementation of user specific models by full integration into the GUI. New implemented models are available after compiling the associated object.

The next planned steps include the implementation of additional models (e. g., such as connected diesel, wind and PV systems [20]) and the improvement of the provided models, e. g., for statistical weather simulation or residential demand. Further efforts will be done to reduce the implementation effort of algorithms for power system simulation. Improvements of usability are a permanent aim within the software project.

References

[1] M. D. Ilic, "From Hierarchical to Open Access Electric Power Systems," *Proceedings of the IEEE*, vol. 95, pp. 1060–1084, May 2007.

[2] P. Palensky, E. Widl, A. Elsheikh, and S. Member, "Simulating Cyber-Physical Energy Systems: Challenges, Tools and Methods," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, pp. 318–326, Mar. 2014.

[3] M. Pöchacker, A. Sobe, and W. Elmenreich, "Simulating the smart grid," in *IEEE PowerTech*, (Grenoble, France), June 2013.

[4] M. Pöchacker, T. Khatib, and W. Elmenreich, "The Microgrid Simulation Tool RAPSim: Description and Case Study," in *IEEE Proceedings of ISGT Asia 2014*, (Kuala Lumpur), pp. 287–292, IEEE, May 2014.

[5] R. D. Zimmerman, C. E. Murillo-Sanchez, R. J. Thomas, C. E. Murillo-s, and R. J. Thomas, "MATPOWER's extensible optimal power flow architecture," in *2009 IEEE Power & Energy Society General Meeting*, no. x, pp. 1–7, IEEE, July 2009.

[6] C. A. Canizares and F. Alvarado, "UWPFLOW: continuation and direct methods to locate fold bifurcations in AC/DC/FACTS power systems," *University of Waterloo*, 1999.

[7] H. Bindner, O. Gehrke, P. Lundsager, J. C. Hansen, and T. Cronin, "IPSYS–A simulation tool for performance assessment and controller development of integrated power system distributed renewable energy generated and storage," *WREC VIII, Denver, Colorado*, 2004.

[8] S. Cole and R. Belmans, "MatDyn, A New Matlab-Based Toolbox for Power System Dynamic Simulation," *IEEE Transactions on Power Systems*, vol. 26, pp. 1129–1136, Aug. 2011.

[9] K. Anderson, J. Du, A. Narayan, and A. E. Gamal, "GridSpice: A distributed simulation platform for the smart grid," *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, vol. 3203, no. c, pp. 1–5, 2013.

[10] D. P. Chassin, K. Schneider, C. Gerkensmeyer, S. Member, A. W. Gridlab-d, K. Schneider, C. Gerkensmeyer, and A. W. Gridlab-d, "GridLAB-D: An Open-source Power Systems Modeling and Simulation Environment," in *2008 IEEE/PES Transmission and Distribution Conference and Exposition*, pp. 1–5, IEEE, Apr. 2008.

[11] E. Widl, P. Palensky, and A. Elsheikh, "Evaluation of two approaches for simulating cyber-physical energy systems," *38th Ann. Conf. on IEEE Industrial Electronics Society (IECON)*, pp. 3582–3587, Oct. 2012.

[12] S. Schütte, S. Scherfke, and M. Sonnenschein, "Mosaik - Smart Grid Simulation API - Toward a Semantic based Standard for Interchanging Smart Grid Simulations.," in *SMARTGREENS*, pp. 14–24, 2012.

[13] M. D. Ilic, U. A. Khan, and M. D. Ili, "Modeling future cyber-physical energy systems," in *2008 IEEE Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century*, pp. 1–9, IEEE, July 2008.

[14] P. Oliveira, T. Pinto, H. Morais, and Z. Vale, "MASGriP - A Multi-Agent Smart Grid Simulation Platform," 2012.

[15] T. K. Wijaya, D. Banerjee, T. Ganu, D. Chakraborty, S. Battacharya, T. Papaioannou, D. P. Seetharam, and K. Aberer, "DRSim: A cyber physical simulator for Demand Response systems," in *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pp. 217–222, IEEE, Oct. 2013.

[16] A. Sobe and W. Elmenreich, "Smart Microgrids: Overview and Outlook," in *Proceedings of the GI INFORMATIK Workshop on Smart Grids*, no. 1, (Braunschweig), 2012.

[17] A. Monacchi, D. Egarter, W. Elmenreich, S. D'Alessandro, and A. M. Tonello, "GREEND: An energy consumption dataset of households in italy and austria," in *Proc. IEEE International Conference on Smart Grid Communications (SmartGridComm'14)*, (Venice, Italy), 2014.

[18] S. Gottwalt, W. Ketter, C. Block, J. Collins, and C. Weinhardt, "Demand side management - A simulation of household behavior under variable prices," *Energy Policy*, vol. 39, pp. 8163–8174, Dec. 2011.

[19] A.-G. Paetz, T. Kaschub, P. Jochem, and W. Fichtner, "Load-shifting potentials in households including electric mobility - A comparison of user behaviour with modelling results," in *10th International Conference on the European Energy Market (EEM)*, pp. 1–7, IEEE, May 2013.

[20] T. Khatib and W. Elmenreich, "Novel simplified hourly energy flow models for photovoltaic power systems," *Energy Conversion and Management*, vol. 79, pp. 441–448, Mar. 2014.