MODAM: A MODular Agent-Based Modelling Framework

Fanny Boulaire
Mark Utting
Robin Drogemuller
Faculty of Creative Industries
Queensland University of Technology
Brisbane, Australia

***Abstract*— Designing the smart grid requires combining varied models. As their number increases, so does the complexity of the software. Having a well thought architecture for the software then becomes crucial. This paper presents MODAM, a framework designed to combine agent-based models in a flexible and extensible manner, using well known software engineering design solutions (OSGI specification [1] and Eclipse plugins [2]). Details on how to build a modular agent-based model for the smart grid are given in this paper, illustrated by an example for a small network.**

***Index Terms*— Modular, Agent-based model, OSGi**

I. Introduction

The future grid is going to be smart, where information and communication technology will allow the automation of the delivery of electricity in an efficient, reliable and sustainable manner [3]. To plan such a grid, it is necessary to select the appropriate components that are able to communicate useful information, to understand their most appropriate placement and finally to tune them so that they can be used in their most efficient manner. These three phases have been considered in a large project the work presented in this paper belongs to [4]. Modelling and simulation has been used in this project to represent the network and trial scenarios to understand its possible evolution; combining agent-based modelling and particle swarm optimisation to model the overall distribution grid.

The work presented in this paper concentrates on the specifics of the agent-based model, implemented in a modular manner. Agent-based modelling has been chosen to model the design of the smart grid for its capacity to describe the components at various scales (defining the granularity of the model) and their behaviour (using decision-making heuristics, learning rules or adaptive processes which make them autonomous). The contribution of this paper lies in the fact that this agent-based model is built in a modular manner, taking advantage of good software engineering practices to build extensible and flexible software. That way, models can evolve as the information relating to the smart grid changes or becomes available or more accurate. Also, models can be implemented by different groups for different analysis types and combined to obtain a more complete system representation. For this, the modules need to follow the guidelines described in this paper, implemented in Java. The MODAM (MODular Agent-based Modelling) framework gives the backbone structure of the modular implementation of ABMs, allowing modules to communicate via named data values and user-defined Java interfaces.

In the first part of this paper, the different challenges faced when building a simulation tool for a smart grid are presented, along with the justification for using an agent-based model and building the software in a modular manner. Details on the architecture of the modular agent-based model (MODAM) follow. The different concepts introduced are illustrated using an example for a small network with solar panels.

II. Why use a modular agent-based model to describe the smart grid?

This section places the context of the work presented here, followed by the justification in using agent-based modelling to represent the smart grid. It then describes the solution in implementing it using a modular approach.

A. Context of the project and challenges from studying such a grid

The scope of the project is to understand and model the future electrical distribution network managed by Ergon Energy, one of the two electricity distribution companies that serve the state of Queensland, Australia. Ergon's distribution network covers 1.7 million square kilometres and provides power to approximately 650,000 homes and businesses across regional and rural Queensland. The network consists of approximately 150,000 kilometres of power lines and associated distribution infrastructure. Ergon also owns and operates 33 stand-alone power stations that power isolated communities across Queensland that are not connected to the main electricity grid. Two types of network are used: 3 phase network as well as SWER (Single Wired Earth Return). The consumers are of different types (residential, commercial, and industrial) and can also be producers via the use of decentralised generators (solar panels, batteries…).

Having described this system as such, it becomes clear that modelling such a grid poses challenges in terms of modelling techniques and software implementation constraints. Table 1 highlights a few of the challenges when modelling the smart grid, and how they translate in terms of the modelling and software implementation requirements (data and modelling).

TABLE 1 – SOME CHALLENGES OF THE SMART GRID IN THE CONTEXT OF OUR PROJECT, AND THEIR TRANSLATION IN TERMS OF MODELLING REQUIREMENTS.

Challenges of the Smart Grid	Technical constraint/Geographical scale • Different technical systems ○ 3 phase systems vs. SWER networks • Distance constraints between components – network density variability • Variation in terms of usage and load types Behaviour of the system • From centralised to decentralised ○ Producers and Consumers at same location ○ Bi-directional flow of information • Changes in terms of usage behaviour ○ Incentives for usage at different times from time of use tariffs ○ Changes in net usage behaviour from decentralised generators (PV, battery…) Technical - feedback loops • Higher response from the controllers to changes in consumptions Feedback loop and autonomous response
Data and Modelling requirements	Data requirements • Different types of information ○ Different data for the network topology (SWER, 3 phase) ○ qualitative vs. quantitative information ○ Different time scales (records every 1/2 hour, 5 minutes…) • Different databases holding the information • Large datasets to manage and analyse Modelling/Analysis requirements • Need a good representation of the system actors ○ Capture individual behaviours ○ Capture interactions of behaviours • Large variation – no averages

B. A technical solution to implementing a smart-grid simulation tool

From this list of data and modelling requirements, a modular agent-based model was chosen as the solution to represent the smart grid. Reasons for such a choice are given below.

1) The use of agent-based modelling to model the smart grid

According to House-Peters [5], the popularity of ABMs for the analysis of complex systems is mainly due to their ability: "to incorporate both spatially and temporally explicit data, to model bidirectional relations between individual human agents and the macrobehavior of the social or environmental system being modeled, to capture emerging patterns at higher scales of the system that result from interactions at lower levels, and to blend qualitative and quantitative approaches". These are some of the reasons agent-based modelling was chosen for modelling the smart grid, as they are answering some of the challenges described in Table 1 (varying types of data - spatial and time, individual behaviours and interactions, qualitative and quantitative information).

Other considerations for choosing agent-based modelling were that agent-based modelling can model individual components and their individual behaviour which other modelling techniques cannot. For example, statistical techniques which are based on analysing large samples of individuals of similar behaviour cannot capture appropriately networks that are sparsely populated, and for which electricity usage can differ greatly. Indeed, SWER networks often have 30km between each residence with only 20 or 30 customers under a feeder, some of which can be farmers with very different behaviours. The sudden changes in electricity needs, when a farmer starts irrigating, are greatly affecting the distribution compared to having many users requiring small loads but in a more even manner.

Finally, one of the interest in using agent-based modelling for the smart grid is the capability of the agents to be autonomous, to have explicit goals that drive their behaviour and to learn from past experiences [6]. This is an important quality as its components are to automatically adapt to the changes in the network and respond to them so that electricity is delivered in an efficient manner. Rule-based or evolutionary algorithms, for example, can be trialled in the simulation tool before being implemented on the grid.

2) Designing the software in a modular manner

Studying the smart grid requires building a system that can be rather large in terms of the number of components that need to be modelled as well as the representation of their interactions. It also necessitates many different types of analyses depending on which aspect of the smart grid a user is interested in.

Consequently, building software for the smart grid will lead to large software systems which will become more and more complex as they are being built. One way of avoiding complex and difficult to extend and maintain software, is to build them in a modular manner. "*Modularity involves breaking a large system into separate physical entities that ultimately makes the system easier to understand. By understanding the behaviours contained within a module and the dependencies that exist between modules, it's easier to identify and assess the ramification of change*" [7]. That way, not only is it easier to build the system, it is also easier to modify parts of the code when more is learnt about smart grids as they develop.

Also, from Table 1, having different databases holding the information was identified as a challenge to modelling the grid. Being able to handle all these data types is made easier by having a modular approach, where each element of the software system can use one or many data formats, and common interfaces are used to hide the differences between alternative suppliers of similar data.

III. Related work

Modularity is not a new concept. Parnas in [8] describes it as information hiding, where separation of concerns [9] are respected. Many programming languages have been developed with this concept in mind, with object-oriented programming one of them. While modularity can be seen at the object level, such as with the object-oriented paradigm, the modularity can happen at a higher level of granularity. For example, component-based software engineering [10] is an example of a modular implementation at the software level. The work presented in this context considers modularity at the software level, i.e. at the component level. However, it also uses modularity at a fine level through the use of agents that are defined in an agent-based model. Unlike EPOCHS [11], our agents are the finest level of granularity of the system, and contain the information describing their behavior through their implementation; an EPOCHS 'agent would be what we call a module. More detail on modularity in the MODAM context is given below.

IV. The framework

Having demonstrated the interest of implementing an agent-based model in a modular manner for the simulation of a smart grid, this section describes how such a framework can be built, and what software engineering solutions have been used for this. An example of an implementation is given here to illustrate the principles described; it describes a network containing solar panels. The example describes the different parts of the simulation set up but does not go into details about the whole simulation; more details on the whole simulation can be requested to the authors.

A. Technology - OSGi and Eclipse plugins

The implementation of the platform was done in Java using the Eclipse Platform [2] . The modularity of the platform was achieved through the use of Eclipse plugins which can be defined as OSGi bundles. OSGi is a specification as defined by the OSGi Alliance (formerly Open Services Gateway Initiative) [1], and OSGI bundles are defined as the unit of modularisation [12]. A bundle is a self-contained unit which explicitly defines its dependencies on other modules and services, as well as its external API.

The notion of modularity is not specific to any technology, and can be achieved using principles in standard Java, for example [13]. However, OSGi offers a higher value solution to the problem of encapsulation, and eases the modularity of a software: "*This is where a module framework, such as OSGi, shines because it allows you to carefully encapsulate implementation details within a module through its explicit import package and export package manifest headers*" [7], chapter 3. Consequently, the programmer can effectively control the provided API and the dependencies of their plugins. The modules and services can also be dynamically activated, de-activated, updated and de-installed, which makes their use very flexible, especially since it is possible to change the configuration of the system at runtime. For example, it is possible to download a module from a website and add it to a model without recompiling or recoding any module.

B. MODAM Framework

The MODAM (MODular Agent Model) framework is the backbone structure for a modular agent-based model implementation. It is currently only used in the context of the smart grid; however, it is not limited to it and is applicable to other domains such as the water or transport areas. Consequently, some definitions below are quite generic, while examples are always referring to the smart grid.

1) Breakdown of the software into reusable modules

A module in the MODAM framework is defined as:

$$Module = Name + Assets + Agents + Data \qquad (1)$$

A module is here an Eclipse plugin. As described above, OSGi was the chosen technology, and the framework is implemented using Eclipse plugins as the development unit. Plugins are the smallest deployable and installable software components of Eclipse, and they can define extension points, which allows other plugins to reference and use them.

Figure 1below shows an example of 2 modules, along with the definition of the extensions from the framework plugin. This example will be used all along the paper to support the description of the different components making MODAM.

We can see here that the framework plugin defines 3 extensions: Data Provider, Asset Factory and Agent Factory. These are used in any plugin that extends the framework through their plugin.xml file. In Figure 1, PvAsset plugin has a class that extends the Asset Factory, and PvAsset.reader one that extends the Data Provider; the two modules are also linked through the use of data values. This way, each module satisfies the definition above.

One important feature of this framework is the use of the factory pattern. Assets and agents in a module are not created manually and individually; they are created using factories (implementing the interfaces Asset Factory and Agent Factory). This process is done in an automated manner, using information held by the data providers (more details on this below). This way of creating the agents differ from the classic agent-based model implementations such as Repast [14] or MASON [15] which require the modeller to define agents individually. Here, the agent factory will create all the agents it finds, from the assets already created, not just some specified. Also, binding the module to its data is a requirement of this framework, which is done through the data providers (more detail on this below).

2) Defining the interfaces from the modularity requirements – extension points

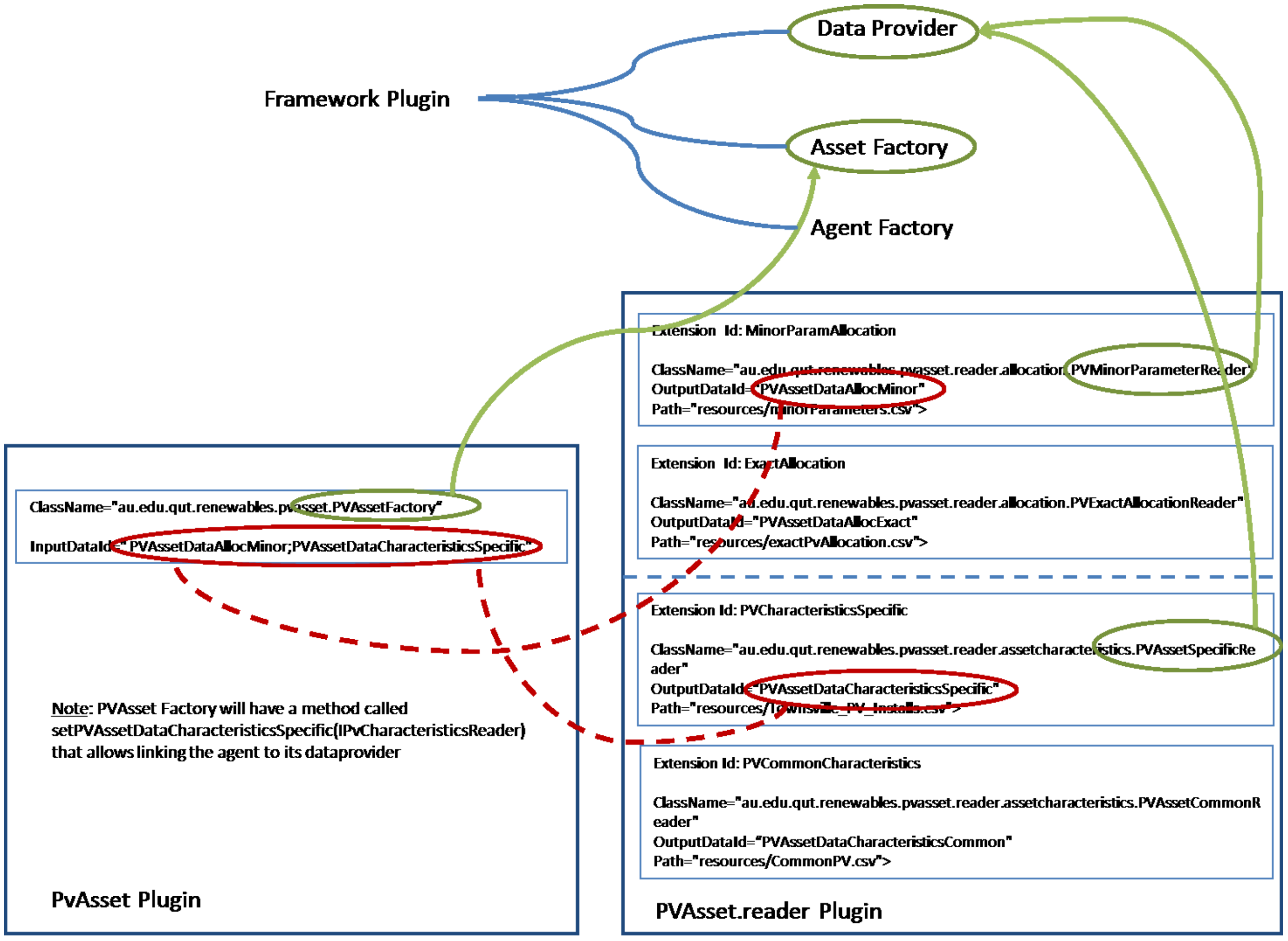
Modularity within the Agent-Based Model - Separation of asset and agents

One of the first steps towards modularity was the identification of 2 main types of entities in the definition of the model: assets and agents. The assets are the entities that describe the network topology (information about their characteristics and their physical connections) and the agents describe their behaviour. In a classic way, an agent would normally contain both the characteristics and the behaviour in one single object. Here a clear distinction has been made between the assets and the agents, giving more flexibility to the model. For example, the behaviour can be defined as the consequence of a given policy. Having many policies to be tested, these can be implemented in the behaviour and easily tried by assigning them to an asset without needing to modify one or more of the methods of the object.

Also, using this approach, an asset can be assigned 1 or more agents to describe its behaviour, allowing an asset to remain the same even if its user changes its behaviour during its lifetime. An example of this would be of a premise asset that would see a change of tenants and consequently of electricity usage, while still maintaining the same characteristics in terms of insulation when calculating the heating and cooling needs of the building envelope.

Modularity when populating the model – using different data readers

In order to answer the challenge of dealing with different databases that hold the information, two approaches at least can be taken. One is to implement one type of readers in the software, requiring data manipulation before importing the file. Another is to have a different reader for each of the databases, so that file types are dealt with individually to populate the model. The second option was chosen in the MODAM implementation as it offers more flexibility; the first



```
ClassName="au.edu.qut.renewables.pvasset.PVAssetFactory"

InputDataId=" PVAssetDataAllocMinor;PVAssetDataCharacteristicsSpecific"
```

Note: PVAsset Factory will have a method called setPVAssetDataCharacteristicsSpecific(IPvCharacteristicsReader) that allows linking the agent to its dataprovider

```
Extension  Id: MinorParamAllocation

ClassName="au.edu.qut.renewables.pvasset.reader.allocation.PVMinorParameterReader"
OutputDataId="PVAssetDataAllocMinor"
Path="resources/minorParameters.csv">

Extension  Id: ExactAllocation

ClassName="au.edu.qut.renewables.pvasset.reader.allocation.PVExactAllocationReader"
OutputDataId="PVAssetDataAllocExact"
Path="resources/exactPvAllocation.csv">

Extension Id: PVCharacteristicsSpecific

ClassName="au.edu.qut.renewables.pvasset.reader.assetcharacteristics.PVAssetSpecificRe
ader"
OutputDataId="PVAssetDataCharacteristicsSpecific"
Path="resources/Townsville_PV_Installs.csv">

Extension Id: PVCommonCharacteristics

ClassName="au.edu.qut.renewables.pvasset.reader.assetcharacteristics.PVAssetCommonR
eader"
OutputDataId="PVAssetDataCharacteristicsCommon"
Path="resources/CommonPV.csv">
```

Figure 1 – Example of definition of 2 Plugins, and their relationship to one another.

PvAsset Plugin contains an extension point for the asset factory, and PvAsset.reader plugin contains many extension points for the data providers. These are shown with the green arrows, showing an implementation of the interfaces defined in the MODAM framework. These two plugins are linked through their InputDataId and OutputDataId – shown in the red dashed lines. Many extension points can be defined in the plugin.xml, and only used when needed through their linking.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
  <extension-point id="dataprovider" name="Data Provider" schema="schema/datareader.exsd"/>
  <extension-point id="agentfactory" name="Agent Factory" schema="schema/agentfactory.exsd"/>
  <extension-point id="assetfactory" name="Asset Factory" schema="schema/assetfactory.exsd"/>
</plugin>
```

Figure 2 – Extension points for the MODAM framework – extract from the plugin.xml file.

one is still possible however. The extension point for Data Provider allows the flexibility to having different data formats as many implementations of a data provider can be done. The object needing access to the data can then call the interface without having to know anything about the data format. If required, the data provider implementation can be switched over without impacting the rest of the code.

Connecting modules together through interfaces

In Eclipse, plugins are connected together through extension points and extensions. The extension points defined in MODAM are given in Figure 2, following the 3 requirements described above (separation of asset and agent, and different data readers). These extension points define interfaces in the code, allowing the different modules to be connected to one another, ensuring the functioning of the modular platform. The definitions of these extensions are given in TABLE 2, where the attributes are described, and the interfaces are given in the *ClassName* attribute of the extensions. These interfaces differentiate this approach from the traditional tools used for agent-based modelling, such as MASON and Repast [16], which are non-modular and combine the data and behaviour aspects of agents.

It has to be noted that additional extensions can be defined if required, and this can be done in the users' modules too. This ensures the extensibility of the MODAM framework, allowing users to build on the existing code.

3) Connecting the modules together when setting up a simulation

The module Manager

Having defined the different elements of the code in the different plugins, these need to be connected to one another. This is done by the module manager that can be seen as the central point of code. Figure 3 shows the lifecycle of the simulation tool. Two parts are distinguished here; the one setting up the simulation (Module Manager) and the one running it (ABM state). The role of the Module Manager is to find all the plugins that are available in the registry, and enable those that have been chosen by the user. From these enabled plugins, the required plugins that are missing are found and added to the simulation.

The module manager can then call the asset factories and the agent factories that have been passed through the extension points of the enabled plugins. Methods are then called on these factories, using reflection, by just knowing the type of interface they implement – these have been passed in through the extension point definition.

Once all the modules have been enabled, the simulation can be started (ABM state part of the lifecycle). The simulation can be started and stop at any time; the simulation is running when the *step()* method of the agents is called upon.

Communication between plugins

Through these extension definitions, the plugins can be linked not only by extending the interfaces, but through the data values. These data values are defined in the plugin.xml file under the *InputDataId* and the *OutputDataId* attributes, and need to have the same value to be linked to one another.

For example, in Figure 1, PvAssetFactory will use the data provider *PVAssetSpecificReader* that has been linked to it through the *PVAssetDataCharacteristicsSpecific* value in the PVAsset.reader plugin; and the data provider *PVMinorParameterReader* linked through *PVAssetDataAllocMinor*. These links are shown in Figure 1

TABLE 2 –MODAM EXTENSION DEFINITIONS.

Asset Factory	Agent Factory	Data Provider
extension	extension	extension
Sequence (1 - *)	Sequence (1 - *)	Sequence
AssetFactoryClass	AgentFactoryClass	point
point	point	id
id	id	name
name	name	DataReader
AssetFactoryClass	AgentFactoryClass	ClassName
ClassName	ClassName	OutputDataId
InputDataId	InputDataId	DefaultPath
Predecessors		

through the dashed lines. These 2 extension points are used in the code to describe the types of solar panels that need to be created *(PVAssetDataCharacteristicsSpecific),* and to which asset they are associated with, i.e. premise or substation number (*PVAssetDataAllocMinor*).

It can be seen as well in Figure 1 that there are 4 extension points for the PvAsset.Reader plugin, but only 2 are used. This demonstrates the flexibility in the model implementation, as different combinations can be chosen to create the solar panels, e.g; it is therefore possible to change the behaviour of the software in an easy way.

While these plugins are linked through the plugin.xml file, their use in the code is automated through the use of reflection in the Module Manager. Consequently, the user does not need to modify the Module Manager, but simply define these 2 attributes with the same value in the plugins and implement a method named "set + ValueOf(*OutputDataId*)" in the factory class. For example, according to Figure 1, the class PVassetFactory then requires 2 methods that will be:

- setPVAssetDataAllocMinor (IDataProvider)
- setPVAssetDataCharacteristicsSpecific (IDataProvider)

Finally, an *InputDataId* can have many values as shown in Figure 1, which are separated by semi-colons.

4) *Cross-connections amongst plugins*

Parameters tracking

As in many software systems, parameters can be set and used at different stages of the simulation. MODAM considers two types of parameters: global and plugin specific. As its name indicates, the plugin specific parameters will be held at the plugin level and cannot be accessed from other plugins. An example of this would be whether the power flow analysis is done using simple or complex power in the power flow plugin. The global parameters however, are defined in the module manager and can be tracked all along the simulation from any plugin. Such parameters are the start time, end time, and random seed.

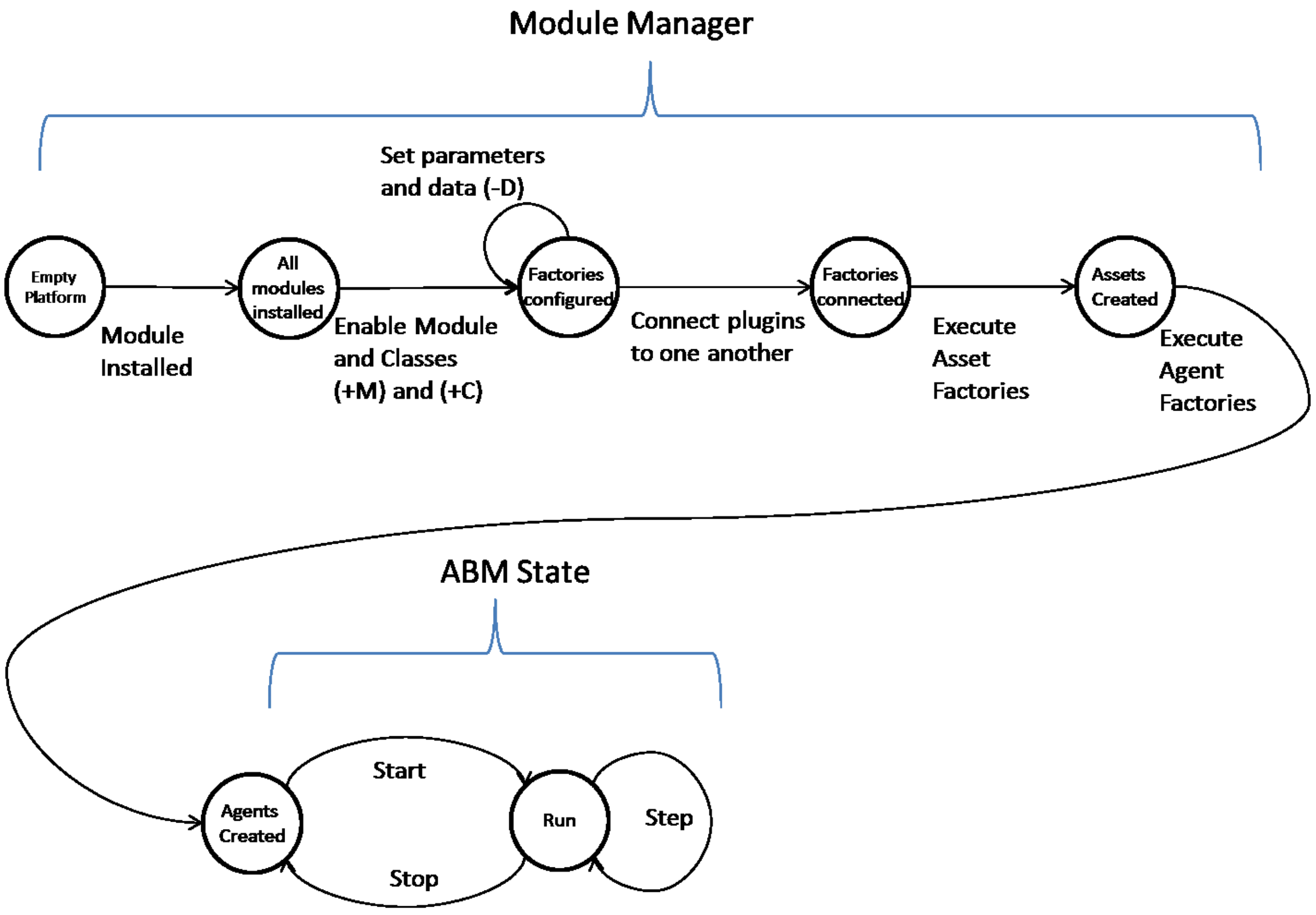


Figure 3 – Illustration of the lifecycle of the simulation tool.

2 phases can be distinguished here: the set-up of the simulation (Module Manager part) and its running phase (ABM state). The Module Manager is responsible for finding all the modules, connecting them together and creating the assets and agents through their factories; data is used to populate them. The simulation can then be started and stopped as required.

Sharing an asset amongst plugins

An asset can be used by different modules by having an agent in each of these modules. For each agent type, different attributes of the asset would be used that can be defined at runtime through the use of channels. Channels are defined here as a set of attributes defined at runtime for an agent. Depending on the value of the channel parameter, the behaviour of the agent is different, through its connection to a different demand data type (residential, commercial data…) or different logic (simple and complex power). While the choice of the parameter is defined at runtime and within a plugin, all the available channels are defined globally and held in the MODAM framework.

The data structure chosen for the channels is a map object where a given parameter will be assigned a value which can be accessed anywhere. This map is located at the ABMState level. The reason for doing this was so that someone who needs to use the MODAM framework in the future will have the mechanisms to use global variables in that manner.

5) Setting up the simulation – command line arguments

In order to run a simulation, modules relevant to the specific analysis can be loaded. For this, a command line reader was created; an example of it is given in TABLE 3.

First, the modules and their classes required to set up the simulation are given, using "+M" and "+C" respectively followed by the names of the required modules and classes. Specifying the classes is not always required, for example when only one type of factories is available in the module. In that case, only the "+M" command will be called. However, if there are different factories in one module for example to describe the network, e.g. network data and SWER data, a distinction can be made as to which needs to be called. Each of the specified classes can also be parameterised using the "-D" command followed by the parameter value. This is then called by the class as an argument and using reflection on the parameter name. For example, '-D = AllocationMethod = "R"' will be used in the specified class with the method *setAllocationMethod (String R)*. Finally, other parameters for the simulation run can be passed. These are the start and end times of the simulation, called using "-from" and "-to". And a folder that will contain the output of the simulation can be specified using "-output".

While many modules can be created, not all of them need to be loaded, only those required for a given analysis type. However, as the model grows, many modules might be required to be loaded as they will ensure that the whole of the system is taken into account. This might lead to a very long command line. To prevent this, and also build on previous simulation runs, it is possible use a configuration file that has been saved in a previous simulation. An example of this is also given in Table 3, which calls "-config" with the name of the file, and adds the new modules that are required for this simulation.

To summarise, Table 3 shows the command line for 2 simulation runs. The first one is to run the demand on a network and uses information from 3 plugins which create the assets and the agents using the data provided by the readers. The simulation is to be run for 1 week, from the 01/01/2010 until the 08/01/2010, and the output of the simulation will be saved in the tempOutDir directory. The second simulation builds on this one (calling the network.xml file), and 4 additional modules are loaded. These 4 modules are for the modelling of the solar panels.

I. APPLICATION OF MODAM

Using the approach described above, many simulations have been performed, investigating different parts of the system. For example, in addition to the 3 phase network that is mostly found in cities, simulations on a SWER network were performed to study the load variations on a rural network in central Queensland. The voltage drops seen by each customer as the load varies were calculated using a load flow analysis. A battery plugin was added where battery assets could be placed on the network to support voltage drop at places under stress. By adding other plugins that describe the batteries behaviour, different control algorithms could be tried to identify the ones that would be most helpful to the network.

When assessing the impact of renewables on the grid, and more particularly describing the behaviour of solar panels, many different approaches can be taken. One is to use historical data for given solar panels and reuse them in future years, expecting simular weather output. Another one is to simulate the PV output using weather information as well as

TABLE 3 – EXAMPLE OF 2 SIMULATIONS SET UP, USING COMMAND LINE AND CONFIGURATION.

Command Line example for network simulation	Reuse of an existing configuration file plus additional commands
+M= assetreader	-config=network.xml
+C=assetreader.NetworkReader	+M=pvasset
+C=assetreader.LocationReader	+C=pvasset.PVAssetFactory
+M=demandreader	+M=pvagent
+C=demandreader.historical.HistoricalDemandReader	+C=pvagent.WeatherPVAgentFactory
+C=demandreader.billing.BillingDataReader	+M=pvasset.reader
+M=assetnetwork	+C=pvasset.reader.allocation.PVMinorParameterReader
+C=assetnetwork.ergon.NetworkAssetFactory	+C=pvasset.reader.assetcharacteristics.PVAssetCommonReader
+C=assetnetwork.agent.NetworkAgentFactory	+M=weatherreader
-from=2010-01-01 -to=2010-01-08	+C=weatherreader.CloudDataReader
-output=tempOutDir	+C=weatherreader.TemperatureDataReader
	-output=tempOutDir

the usual physical equations, and predict the PV output taking into account the passage of clouds – details on this implementation can be found in [17]. These 2 approaches have been implemented in 2 distinct plugins and can be selected indifferently depending on the needs of the user.

With time, it is expected that many more plugins will be added so that the behaviour of the system can be captured in its entirety. One of the near future tasks is to explore different types of demand-side management (DSM) and their uptake level on the grid. Each of these DSM options is expected to be implemented in separate plugins, and called at setup of the scenario depending on the type of assessment required. Most of the power grid scenarios are handled by adding new modules and/or extending the existing modules to have flags and parameters to give more control over their behaviour.

II. Conclusion

Smart grids can be modelled using agent-based modelling in a modular manner which is an efficient manner of building software. Taking such an approach allows building on previous work, as the simulation environment grows and more data becomes available. This paper demonstrated that such an approach is possible through the illustration of the implementation of the functionalities on a network with solar panels. The code for the MODAM framework which is open-source can be used for the implementation of user functionalities of the smart grid as more data become available. Examples of functionalities of the smart grid, such as feedback loops haven't been shown here, because the aim of this paper was rather to set the architecture for a modular approach to agent-based modelling in the view of simulating the smart grid rather than the different algorithms that populate the software. More details on the implementations of the functionalities will be given in a later paper. This paper showed that current software engineering techniques can be useful in developing solid software for the smart grid.

III. References

[1] OSGI Alliance. (2013, 01/02/2013). *OSGI Alliance*. Available: http://www.osgi.org/Main/HomePage

[2] The Eclipse Foundation. (2012, 27/02/2012). *About the Eclipse Foundation*. Available: http://www.eclipse.org/org/

[3] The National Institute of Standards and Technology (NIST). (2012, 06/02/2013). *Smart Grid: a Beginner's Guide* Available: http://www.nist.gov/smartgrid/beginnersguide.cfm

[4] F. A. Boulaire, M. Utting, R. Drogemuller, G. Ledwich, and I. Ziari, "A Hybrid Simulation Framework to Assess the Impact of Renewable Generators on a Distribution Network," in *2012 Winter Simulation Conference*, Berlin, Germany, 2012.

[5] L. A. House-Peters and H. Chang, "Urban Water Demand Modeling: Review of Concepts, Methods, and Organizing Principles," *Water Resources Research,* vol. 47, p. W05401, 2011.

[6] C. M. Macal and M. J. North, "Agent-Based Modeling And Simulation," in *2005 Winter Simulation Conference*, 2005.

[7] K. Knoernschild, *Java Application Architecture: Modularity Patterns with Examples Using OSGi*: Prentice Hall, 2012.

[8] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM,* vol. 15, pp. 1053-1058, 1972.

[9] E. W. Dijkstra, "EWD 447: On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective,* pp. 60-66, 1982.

[10] C. Szyperski, *Component software: beyond object-oriented programming*. New York: ACM Press, 1997.

[11] K. Hopkinson, W. Xiaoru, R. Giovanini, J. Thorp, K. Birman, and D. Coury, "EPOCHS: a platform for agent-based electric power and communication simulation built from commercial off-the-shelf components," pp. 548-558.

[12] L. Vogel. (2012, 24/09/2012). *OSGi Modularity - Tutorial*. Available: http://www.vogella.com/articles/OSGi/article.html

[13] K. Knoernschild. (2012, 04/12/2012). Patterns of Modular Architecture. *DZone Refcardz,* 7. Available: www.dzone.com

[14] Argonne National Laboratory. (2011, 25/09/2011). *Repast Simphony*. Available: http://repast.sourceforge.net/repast_simphony.html

[15] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A Multi-Agent Simulation Environment," *Simulation: Transactions of the society for Modeling and Simulation International,* vol. 82, pp. 517-527, 2005.

[16] C. Nikolai and G. Madey, "Tools of the Trade: A Survey of Various Agent Based Modeling Platforms," *Journal of Artificial Societies and Social Simulation,* vol. 12, p. 2, 2009.

[17] F. A. Boulaire, M. Utting, R. Drogemuller, A. Abeygunawardana, G. Ledwich, and J. Bell, "Planning for the Impact of Distributed Solar Energy on the Grid," presented at the Solar 2012 Conference, Swinburne University of Technology, Melbourne, 2012.