Boulaire *et al. Complex Adaptive Systems Modeling* (2015) 3:1
DOI 10.1186/s40294-015-0007-2

Complex Adaptive Systems Modeling
a SpringerOpen Journal

RESEARCH Open Access

Dynamic agent composition for large-scale agent-based models

Fanny Boulaire[1*], Mark Utting[2,3] and Robin Drogemuller[1]

* Correspondence:
Fanny.Boulaire@qut.edu.au
[1]Queensland University of Technology, Gardens Point – P Block level 8, Brisbane Qld 4000, Australia
Full list of author information is available at the end of the article

Abstract

Purpose: This paper describes *dynamic agent composition*, used to support the development of flexible and extensible large-scale agent-based models (ABMs). This approach was motivated by a need to extend and modify, with ease, an ABM with an underlying networked structure as more information becomes available. Flexibility was also sought after so that simulations are set up with ease, without the need to program.

Methods: The dynamic agent composition approach consists in having agents, whose implementation has been broken into atomic units, come together at runtime to form the complex system representation on which simulations are run. These components capture information at a fine level of detail and provide a vast range of combinations and options for a modeller to create ABMs.

Results: A description of the dynamic agent composition is given in this paper, as well as details about its implementation within MODAM (*MOD*ular *A*gent-based *M*odel), a software framework which is applied to the planning of the electricity distribution network. Illustrations of the implementation of the dynamic agent composition are consequently given for that domain throughout the paper. It is however expected that this approach will be beneficial to other problem domains, especially those with a networked structure, such as water or gas networks.

Conclusions: Dynamic agent composition has many advantages over the way agent-based models are traditionally built for the users, the developers, as well as for agent-based modelling as a scientific approach. Developers can extend the model without the need to access or modify previously written code; they can develop groups of entities independently and add them to those already defined to extend the model. Users can mix-and-match already implemented components to form large-scales ABMs, allowing them to quickly setup simulations and easily compare scenarios without the need to program. The dynamic agent composition provides a natural simulation space over which ABMs of networked structures are represented, facilitating their implementation; and verification and validation of models is facilitated by quickly setting up alternative simulations.

Keywords: Agent-based model; Dynamic composition; Large-scale; Electricity distribution network

Background

Agent-based modelling (ABM) has been used successfully over the last decade to model different aspects of the electricity sector. Its use was initially mainly for the analysis of power market design for large-scale electricity systems when deregulation happened (North et al. 2002; Batten et al. 2006; Weidlich 2008). These models aimed at investigating the interactions between the physical infrastructure at the transmission level (high voltage networks), and the economic behaviour of market participants to help engineer markets in the electricity sector. The application of ABM to the electricity distribution network (medium and low voltage networks) is not as widespread as that of the transmission networks, but is becoming more studied, especially as new technologies (rooftop solar panels, batteries, …) are appearing on the market and transforming the way electricity is consumed, produced and traded (Cai et al. 2011; Institute for Energy and Transport 2014).

Agent-based modelling has seen a rise in popularity for its capacity to provide some insight as to how a system responds to changes from the entities' responses and interactions and the environment, by capturing information at a fine level of detail over space and time using simple rules (Klügl and Bazzan 2012; Macal et al. 2006). It is therefore particularly suited to model the electricity grid which is currently going through a phase of transformation. The way the grid is going to be used is changing, with consumers now also becoming producers and installing new technologies that are changing the flows of electricity on the networks. Communication between the different network assets will become more prominent, impacting further its management but also providing many opportunities. Information about where, how and when these changes are going to happen is important as averages are not sufficient to inform planners appropriately.

Within this context, we have developed a modelling and simulation (M&S) application to answer questions relating to the planning of the future grid and to assess the impact of the integration of decentralised generators (DGs) on a distribution grid owned by Ergon Energy (Ergon Energy 2013). This M&S application supports the network planning process by providing an understanding of the evolution of the network in terms of load and voltages, over space and time, and finding the most economical solution in terms of network upgrades. The full platform uses two modelling techniques: agent-based modelling (Castiglione 2006) and particle swarm optimisation (PSO) (del Valle et al. 2008). The ABM approach was chosen for a few reasons. One is its capacity to capture the information at a fine level of detail both geographically and over time, allowing customers behaviours in relation to their usage of new technologies to be represented and linked to the network structure with accuracy. Another one is that in this application context, the past is no predictor of the future, because very little is known about the impact of the large-scale integration of DGs on distribution networks. By capturing the functioning of the different entities and their relationships to one another, insight into what might happen using simple rules can be gained. In our M&S application, ABM is used to run a large number of scenarios of possible futures, and its output (load duration curves at any node on the network) is passed to the PSO module. This module evaluates which network assets can be installed or upgraded to ensure safe, reliable and economical delivery of electricity. Details on the overall M&S application can be found in (Boulaire et al. 2012b).

This paper focuses solely on the ABM part of the M&S application; more specifically on the technical aspect in building a large-scale agent-based model in regards to the requirements of the project. The following requirements were defined, where the model needed to:

a) Take into account the physical characteristics of the network (assets and their connections) as well as the different actors and the environment, influencing the flows on the network;
b) Be able to represent the evolution of the system over many years (long-term planning) but with a fine level of detail that captures how the different elements operate over half-hourly periods;
c) Deal with large and varied datasets coming from corporate databases to populate the model - in terms of configuration of the network and characteristics of its elements, and also allow for different data types for the output of the simulation;
d) Be able to evolve along with the changes in the network and the consumers market
   a. With the addition of new technologies over time, as they became available;
   b. and different ways in using them, e.g. comparing behavioural or policy impact depending on how the technology is used;
e) Allow creating various scenarios with ease so that simulations can be set up on a daily basis by power engineers, who are not programmers.

Further to these model requirements, goals were identified for its implementation:

- An independent author (a developer that is adding new agents) does not need to modify previously written code;
- The models need to be *assembled* following a "*code-free*" approach, where
  - A model user does not need to read or modify the Java code
    - Behaviours of agents can be added/changed without coding – both at the beginning when setting up the model, and during the run,
    - A user can try different models without going into the code, but simply by combining different aspects of the model
  - There is no need for recompiling when adding agents to the model

Two early implementations of our M&S application, based on Repast (Argonne National Laboratory 2014) and MASON (Luke et al. 2005), exposed various shortfalls with building large-scale models using existing model building approaches in regards to our needs. These are summarised in Table 1. The limitations, requirements and goals mentioned above, led to the development of MODAM (MODular Agent-based Model), a framework that builds flexible and extensible large-scale ABMs, using *dynamic agent composition*. This approach consists in having the agents built at runtime by bringing together the physical representation of the elements (assets) and their different behaviours that will specify their actions.

The solutions implemented in MODAM, in response to the shortfalls identified are also given in Table 1, and are discussed further in the paper.

Table 1 Shortfalls of existing model building approaches and software systems, and MODAM solutions

Shortfalls of existing model building approaches and software systems	MODAM solutions
• Central simulation class is responsible for	• Decentralised factories create the physical properties of an agent (assets) and its behavioural properties (behaviours) separately
a. Instantiating all agents;	
b. Defining relationships between instantiated agents;	
c. Reading the data used to build agents and relationships (if the model is data-driven);	a. Several asset factories create assets and the relationships between them;
d. Complex option handling done as centralised code.	b. Several behaviour factories attach behaviours to assets;
	c. Each asset factory can be independently parameterised with data providers;
	d. Each factory handles its own options.
• Each agent is a single class.	• Separation of assets and behaviours allows
a. Variation of behaviour requires many subclasses;	a. Mix-and-match construction of agents at runtime;
b. Each agent is created independently of others.	b. Gathering certain entities in groups.
• Non deterministic order of agents' execution	• Deterministic simulation runs for each random seed (reproducibility of results)

The MODAM framework enforces the separation of assets and behaviours, and combines this key idea with several other techniques (data providers, factories, interfaces, channels (runtime data attributes), use of reflection by a module manager) to fully support the dynamic composition of agents. Because we are building large-scale ABMs where thousands of agents are represented and for which we have sufficiently accurate knowledge, data is used to populate the model, defining the way the agents are in relation to one another as well as their properties. Assets and behaviours are created within components according to their type, and data providers are called on to populate the individuals' entities, with the model coming together at runtime. This facilitates setting up large-scale simulations and has the additional property of not requiring the user to program. This approach builds on the vision set by (Hamill 2010) of having a library of building blocks for agent-based models. These building blocks would capture a specific environment, or agent, or group of agents and by bringing them together a modeller can set up a simulation more easily, which is especially interesting for the non-programmer. This approach, *dynamic agent composition*, is the key contribution of this paper.

This paper describes this approach, to building flexible and extensible large-scale ABMs. First, the dynamic agent composition is motivated in Section 3 using an example of the implementation of an electric vehicle agent. An overview of the approach is then given in Section 4, followed in Section 5 by more details describing the asset and behaviour models, and a description of how these elements come together at runtime to create a simulation. Section 6 discusses the challenges and the benefits in using this method. Finally, our work is put in relation to other work in the domain of agent-based modelling, and composition.

References to applications of the electricity sector are made throughout this paper to illustrate the use of the *dynamic agent composition* in a concrete manner. More details of the electricity models and simulation results can be found in (Boulaire et al. 2013a; Boulaire et al. 2012a; Boulaire et al. 2012b).

Motivating example of a dynamic agent composition

Before formalising what is meant by *dynamic agent composition*, this section motivates this approach by introducing a simple example of an electric vehicle (EV) agent. This EV agent is to be used within the context of planning the electricity grid, to understand how the increasing number of vehicles and the way they are used will impact the current infrastructure.

An EV is a mobile agent, which is not bound by its geographic characteristics. It can however have properties of location that will indicate where it connects to the grid to charge. It can be considered as a mobile battery, which can be limited to recharge only, but is also able to discharge to support the premise consumption it is attached to, if needed. It has a state of charge which is the amount of energy that is left in the battery, and from which charging requirements are calculated. Therefore it has similar properties to a battery with a few additional ones.

Following from this, an EV has at least two main properties, from the network viewpoint, that influence its behaviour: driving characteristics, and a charging regime. These need to be accounted for when implementing the rules within the EV agent. Figure 1 gives a schematic representation of the composition of an agent, which is made of an electric vehicle asset to which two behaviours are added. The asset includes all the physical and data attributes of the vehicle, such as location, battery capacity, state of charge, maximum charge rate.

While these two behaviours need to be described within the rules, each of them can have multiple implementations. For example, we can think of two implementations for the driving characteristics:

- The location parameter is informed by whether or not the vehicle is at a premise, using a Boolean variable that is set to true for the timesteps after which the EV has reached the premise. This time of arrival can be randomly chosen from a probability distribution curve derived from typical home arrival times. Similarly, its charging state is randomly set from probability distribution curves of typical vehicle trips;
- The location parameter is informed by a traffic simulation system that knows where and when the vehicle has arrived. Its charging state is calculated from the traffic simulation system that knows the exact trip the EV did for the day.

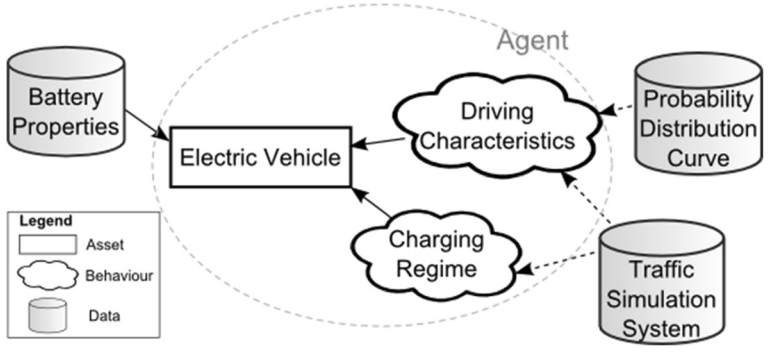

Figure 1 Illustration of the options for an electric vehicle with charging behaviour and driving behaviours.

Similarly, we can also have two implementations for the charging regime:

- The EV can charge at any time of the day, as soon as the signal is set for it (e.g. as soon as the vehicle arrives at the premise);
- The EV charges only during set periods of time, which can be defined by a policy setting (set) or informed from communication with a central controller (dynamic).

For each behaviour type, only two options are presented here, but we can imagine many more alternative behaviours.

If the behaviours and the static information were implemented in one same agent class, these alternative behaviours could be implemented by subclassing an electric vehicle agent class for example, or by calling behaviour objects defined in other classes. In this example, this would result in four agents that the user could choose from, which is equivalent to our implementation. However, when increasing the number of behaviours' options, using the dynamic agent composition would result in lesser implementation needs compared to subclassing existing agents implemented in one class.

Having that flexibility in combining the behaviours is important. Comparison of the impact of behaviours of agents is then facilitated by simply swapping a behaviour with another one.

Further, if a behaviour can be used by two asset types, this separation avoids writing extra code and enables reuse. This is the case for the charging regime in our example, which can be used not only to describe charging characteristics of an electric vehicle but also of a battery, whether it is privately owned or is grid operated. This property of reusability is important as it reduces the development time, and the risk of possible implementation mistakes.

In addition to these properties of flexibility and reusability, having the asset and the behaviours separated also offers ease in extending the model. Indeed, there is no need to modify the code of an agent if a new behaviour type is to be added, even if that behaviour is defined by an independent author. This is quite interesting, especially when more information becomes available as the project evolves, or when information/data changes, e.g. due to new applications.

Finally, data is also used to populate the agents' properties, both for assets and behaviours, which offers additional flexibility in the definition of the agents. For example, projections of EV uptake can be used to create x EV assets in year 1 of the simulation, x' in year 2, etc., and additional data to specify the properties of these assets. Behaviours can be associated to these assets following different percentages of expected behavioural profiles of their users passed in the data (e.g. y% of the population are expected to charge at anytime, and y' % at a set time). These behaviour percentages can also be varied to see the impact an incentive might have on the overall network (e.g. y' could be increased to find the percentage at which users should be encouraged to sign up to a tariff incentive, that would benefit the grid).

These different elements are brought together at runtime, creating agents and the agent-based model through a linking mechanism. This dynamic composition of the agents offers great flexibility and extensibility of the ABM, and means that a modeller does not necessarily need to program; they can just combine assets, behaviours and data to create an ABM. Details are given in the following sections.

Method

Overview of the dynamic agent composition

This section defines the dynamic agent composition and the different types of compositions that are possible, enabling flexibility, extensibility and reusability in the definition of an agent. Then an overview of how a large-scale ABM is built, using this composition, is given.

Definition of dynamic agent composition

We define *dynamic agent composition* as the process of bringing together at runtime the following distinct entities:

- An asset - the physical properties of an agent (static information);
- One or many behaviours - the rules the asset is subject to, to make its decisions (dynamic information);
- Data

where an agent is defined as:

$$Agent = Asset + Behaviours$$

and the data is used to populate either or both the asset and the behaviour attributes. Data is not a requirement, but offers greater flexibility and facilitates the creation of large-scale models. Combinations of these three entities are then held in a component, or module, for their implementation.

Figure 2 shows graphically this dynamic agent composition.

Here, when we mention one asset or one behaviour, we mean one class of asset and behaviour rather than an instance of an asset or behaviour. An asset class will typically have attributes and *setter/getter* methods to populate and access their values, while a behaviour class will contain the rules, held in *start(), step(), stop()* methods.

Data can be used not only to populate the asset and behaviour attributes, but also to determine how many of these are to be created, as well as what their relationship to one another is. Consequently, data reading is not happening at the individual agent level but at a higher level, the factory level, where data is used to populate the individual agents. More detail is given in section 5.

This type of breakdown of the agent into asset and behaviour is especially suited to our domain application, where the physical structure of the distribution network, which is quite static, is to be represented along with the way electricity is consumed or flows over it, which is dynamic. While changes in the infrastructure can happen with upgrades and extension of the network, these are quite slow compared to the dynamic behaviour of

Figure 2 Dynamic agent composition.

the usage of electricity which is the model's state variable, whether it is described as a load, a voltage or a current in the simulations. The assets are likely to be used differently for reasons independent of the asset characteristics, although still within their properties' limits, such as when new policies are implemented that affect user behaviours. Being able to easily change the behaviour facilitates their quick assessment on the system as a whole.

Agent composition types

Many types of agent compositions can be done, which shows the properties of extensibility, flexibility and reusability in building an agent-based model. Figure 3 illustrates many of these compositions.

The base case when extending an agent-based model consists of creating a new agent, which means creating a new asset and its associated behaviours. With the agent composition, an agent-based model can be extended by simply defining new behaviours and applying them to an existing asset, increasing the number of available agent types. This is illustrated with Behaviour B1 for example, which also has alternative implementations (B1', and many up to B1^n).

In addition to extensibility, this example illustrates flexibility, as it is possible to choose any of the available behaviours for a new agent type.

Reusability is illustrated with behaviour B2 which can be used by both Asset A1 and Asset A2. In this case, the two assets have very distinct properties; however, one of their distinct features is that they have in common a set of rules to describe an aspect of their behaviour. This could be for example the case for batteries and electric vehicles, which could both be using the same rules for charging control algorithms.

In these three cases of agent composition, an independent author does not need to modify previously written code. New classes can be created, implementing interfaces to

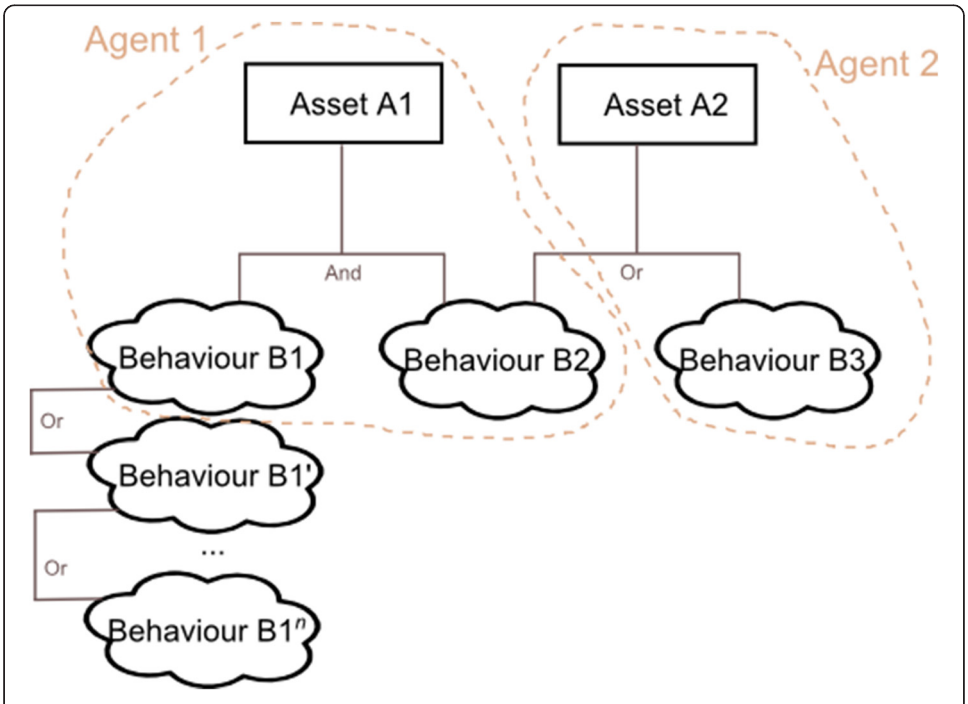

Figure 3 Example of the properties of extensibility, flexibility and reusability when creating agents.

specify whether they are assets or behaviours, and these can then be called at simulation setup to compose the required agent.

Figure 3 shows that an agent is composed of an asset and one or many behaviours. The opposite is also possible, where a behaviour can operate over one or many assets, and update their state at once, within one timestep. Such an example can be found when implementing a global voltage analysis algorithm (load flow (Morton 2003)) which runs over a group of assets, and updates the assets' voltage at each timestep. This is such that the group of assets over which the analysis is happening needs to be balanced, and therefore have a central place of calculation, considering all the assets at once.

Building the agent-based model

When defining an agent-based model, many agents are created and put in relation to one another to form the system over which they will evolve. Using dynamic agent composition, bringing the different assets, behaviours and data together will define a model. Figure 4 illustrates building blocks, or modules, holding these three entities (assets, behaviours and data) where MODAM is the framework that glues them together to form an agent-based model. These modules might contain information relating to many asset or behaviour types at once, or individual ones, depending on the needs. While assets and behaviours are defined individually in their own class, the modules enable them to be grouped together to form sub-systems.

In Figure 4, we can see four modules that contain information relating to assets describing the network: the network assets module (which contains lines, buses, transformers, switches, etc.), solar photovoltaic (PV), battery and EV assets modules. Four modules are also available for the description of the behaviours, where some can contain many behaviours (e.g. the network behaviours), and others share their behaviours across assets (e.g. the battery behaviour for EVs and batteries). Finally, many datasets are represented that inform the assets or the behaviours, and that can be interchanged or used in combination depending on the need of the model or the

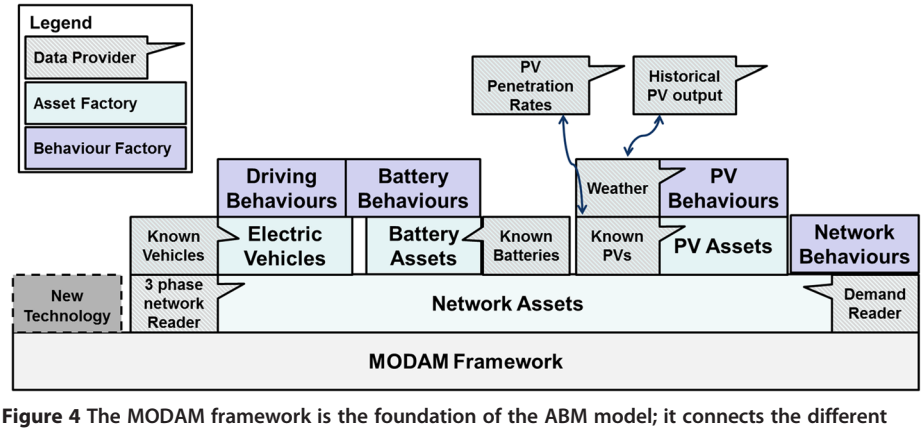

Figure 4 The MODAM framework is the foundation of the ABM model; it connects the different parts of the model. Here a network model describing the network elements and their behaviour is defined. Photovoltaic (PV), electric vehicle (EV) and battery modules have been added to understand how they can support or hinder the functioning of the network. This can be extended in many ways – for example, a 'New Technology' model could be defined and added to the tool to represent any new technology that might impact the network.

availability of the information. For example, for the solar PVs, the assets can be described using known information about their location and their characteristics (that can be used to initialise simulations from the end of the observation period), or using solar PV penetration rates (when making predictions about the future of expected placement and rating of the panels). Similarly, datasets can be used to inform the behaviours, as is the case for solar PV behaviours, which can use a weather model to calculate the output of the solar panels at individual locations taking into account the passage of clouds, or using historical data of solar PV output.

Having described the theoretical framework of the dynamic agent composition, the following section describes how it was implemented.

Results

Implementation of the dynamic agent composition

Our M&S application was developed using the Eclipse IDE (The Eclipse Foundation 2012) which is a widely-used open source platform made of a base workspace and an extensible plug-in system for customizing the environment. Using Eclipse on top of OSGi (Open Service Gateway initiative) and Eclipse plugins, which have strong support for modularity, supported our requirement of a flexible and extensible model environment, and was a natural fit to the definition of our components, or modules, which can each be implemented within their own plugin.

This section describes how our agents are created, using different approaches to the implementation of the assets and behaviours models, and how the data is used to populate them. But first, a UML diagram of the main interfaces used in the MODAM framework is presented in Figure 5; this diagram will be used to support our explanations throughout this section. As a first introduction, an interface named *IABMState* is at the centre of the framework and holds all the elements for a simulation. It sets a scheduler (*IScheduler*), so has access to the simulation time (*ISimTime*), and has access to assets (*IAsset*) which may have one or many behaviours (*IBehaviour*) associated with them. These assets and behaviours are created by factories (*IAssetFactory* and *IBehaviourFactory*) which are populated by data providers (*IDataProvider*). Each asset has a demand object (*IDemand*) that contains an extensible set of named values that can be set; these are the state variables of the simulation. Finally, the scheduler schedules the behaviours which update the demand at each time step during the simulation. More details for each of these entities are given throughout this section.

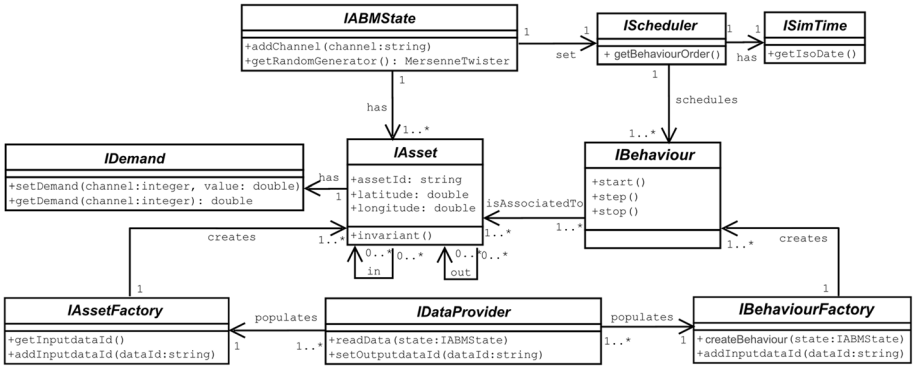

Figure 5 UML diagram of the MODAM framework.

Implementation of an agent

Assets and behaviours are implemented in different ways, according to their requirements.

The assets data model - use of EMF

EMF (Eclipse Modeling Framework (Steinberg et al. 2008)), one of the many available plugins of the Eclipse platform, is a modelling framework that facilitates building code based on structured data models. Particularly well suited to the requirements of our application domain, where the physical infrastructure is to be represented, EMF was chosen to develop a data model describing the assets and their relationship to one another. EMF has the advantage that the implementation of the objects specified in an *ecore* model, described in XMI (XML Metadata Interchange), can have their classes automatically generated, facilitating the implementation of the model within an application.

In addition to this, EMF can handle extension of models, a feature that was of particular interest to us. Any object declared in a data model can be extended or referenced by any other that has been defined in a new child model. Any EMF model can thus be created in a separate plugin and extend one or many models allowing the overall model to keep on expanding. Child models can be in distinct plugins, which makes it possible to choose one model over another at any time, if it describes the problem better, allowing for flexibility in the M&S application.

In our implementation, a few models were defined, which all extend a base model where two entities are defined: *IAsset* and *IDemand,* along with their properties. These entities are implemented as interfaces, as shown in Figure 5. Any other EMF model extending this one can then implement these interfaces, and define others as required. One of the main features of the *IAsset* interface is that it contains two properties (in and out) that define an in-out relationship that specify a directed graph over the assets, to represent the networked structure of our domain problem. In addition, each *IAsset* has a string as a unique ID, as well as a longitude and latitude to give its geographical location. Additional attributes can easily be added to assets, or to particular subclasses of assets, simply by extending the EMF model. This generates new subclasses of *IAsset* that have additional private fields plus getter and setter methods. This is highly customisable, but because it involves code generation and compilation, it is best used for relatively static models and is not sufficiently dynamic to support the kind of runtime composition of behaviours that we want.

Where dynamic composition of behaviours is required, a model designer can use the MODAM *data channels* feature, which is provided by an *IDemand* object associated with each asset. This provides an expandable set of named real-valued attributes for each asset. During the model initialisation phase, the behaviour factories register the channel names that they wish to use, and the MODAM framework maps these to integer indices, so the set of variable channels depends upon which behaviours are included in the model. As the model runs, behaviours can read and write the channel values of any asset. This allows behaviours written by different authors to communicate via data channels simply by using a common name for a channel. It also allows MODAM to provide a generic graphing and logging facility that can graph and save any channels from any set of assets. This can be used for visualisation in Google Earth to provide a platform for stakeholders' engagement.

Two EMF models have been implemented so far in our M&S application that both extend this base model, and have been used together to define the overall asset model. The first model contains the different assets that define a base network, as captured in Figure 4, in the Network Assets module, as well as solar PV and batteries. The second one contains one asset that describes the way a premise would consume electricity depending on the tariff it is subject to. This second model was implemented to test our hypothesis that it is possible to extend the data model using EMF and that this can be done within other plugins. This implementation was successful and the tariff asset was easily added to the model and integrated within the agent-based model.

The behaviours – use of the strategy pattern

Assets and behaviours are implemented independently; however, each behaviour has a reference to its asset in order to retrieve the necessary state variables and make its decisions during a simulation. The implementation of the behaviours' information and rules is contained in the *start()*, *step()* and *stop()* methods of its class that extend an *IBehaviour* interface. The *start()* method belonging to the corresponding asset allows initialising the behaviour, while the *stop()* ends it; the *step()* method updates the behaviour at every timestep.

To obtain flexibility in the behaviour implementation, we used the strategy pattern (Gamma 2009), using interfaces and defining classes that implement the *start(), step(),* and *stop()* methods. Any number of classes can implement the *IBehaviour* interface, and be called at runtime to specify which behaviour is to be used. One advantage of this approach is that a new plugin can easily define new behaviours, as long as access to the interface is provided; there is no need to access a behaviour class previously defined, only the interface.

Building an agent-based model

Building an agent-based model requires bringing together the assets and the behaviours that form agents and relate them to one another to form the complex system over which they will evolve. This is done within factories, using data from corporate datasets, as explained below.

Factories

Assets and behaviours are created separately, which is done automatically through the use of the *factory* pattern. A plugin can contain one or more asset factories that can control which assets need to be created; many factories can be defined for a given type of asset creation for example, with slight variations depending on the aim of the factory. The same is true for the behaviours.

These factories implement the *IAssetFactory* and *IBehaviourFactory,* depending on whether assets or behaviours need to be implemented. Using the *factory* pattern ensures that the action and interactions of the agents are taken care of in a consistent manner.

To answer our goal of flexibility, each factory typically creates assets or behaviours for one specific type of asset or type of behaviour. This means that each asset defined in an EMF model can implement its own *IAssetFactory*; similarly each behaviour type can implement its own *IBehaviourFactory*. It is however also possible to have a factory

that will define many different entities and put them in relation to one another if judged appropriate; this is most likely to happen for the assets only, and is not recommended for the behaviours. An example of this is for the network asset module which contains one factory implementing *IAssetFactory* where lines, buses, switches, transformers and premises are created and related to one another, as assets form the underlying directed graph over which behaviours act.

Factories and data to populate the model

In each of these factories, data can be used to populate the assets and the behaviours. Different types of data can be used, that can define the number of assets to be created, how they are in relation to one another, or what their properties are. One of the requirements of our ABM was that the distribution network be built from corporate data. This means that different types of datasets needed to be handled, and that allowance be made for new types of dataset formats to be input to the simulation. For this, an interface called *IDataProvider* is accessible by both the *IBehaviourFactory* and the *IAssetFactory* during the model instantiation and is implemented by different readers that access various data formats.

Data offers flexibility in setting up simulations as it can be used to represent different areas of study, to compare different trajectories of possible futures by setting different methods of technology uptake, or different methods to describe behaviours. For example a demand behaviour which is associated to a premise and represents its electricity consumption for every half-hour of a day can be set using three types of data in our implementation: half-hourly profile data from a sample of premises, half-hourly profile data from some feeders, and profiles derived from a weather-driven model of consumption. Any of these methods can be chosen to populate the behaviour of a premise consumption individually, and can also be combined using weights to obtain a desired proportion of profile methods.

Depending on the provenance of the data, the format will change, which is handled by different implementations of the same interfaces, providing flexibility in composing the agent-based model. As an example, two types of networks can be used in our current implementation of the agent-based model: a three phase urban network and a SWER (Single Wired Earth Return) network, with data coming from different corporate databases with different formats.

How the ABM comes together

Large-scale agent-based models can be built using dynamic agent composition, either:

- Via explicit Java code, or
- Via command line configuration or GUI, and the use of an automated model builder.

When using explicit Java code, the programmer needs to instantiate the factories, link them to the desired data providers, and execute the factories to build the model. However, this process was automated to answer our goal of code-free construction of agent-based models. This automation is the subject of another paper, but we are giving here an overview of the automation here. A *Module Manager* automatically discovers the

plugins and weaves them together. This means that it finds all the available plugins in the registry and enables those chosen by the user. If there are missing plugins, these are found and added automatically. The assets and behaviours are then created with the required data by the Module Manager who instantiates and parameterises the asset and behaviour factories. Each factory handles its own options that have been given in the command line or GUI panel (parameters, and data). Then methods are called on these factories to populate the assets and behaviours with the required data, using reflection, by just knowing the type of interface they implement.

Discussion

Dynamic agent composition: challenges and responses

While the concept of dynamic agent composition is quite simple, its application to the implementation of large-scale agent-based models with an underlying networked structure has its challenges. These are described below with a discussion about the way we responded to them.

Ordering of the assets' creation

Extending an existing structure, which can be represented by a directed graph, such as the electricity grid where the nodes are assets and the edges their connections, requires a notion of reference. This can be challenging especially as the assets may be defined in separate factories, in separate plugins, and come together to form the complex system only at runtime. For example, if adding battery assets (B), they need to be created and attached to the relevant node (N) (e.g. a premise) in the network. This means that N need to have been created first, and put in relation to the other assets in the initial network. Only then will the battery asset factories be called to create B and attach them to the right nodes N.

To satisfy this requirement of reference over the assets, precedence of the creation of some assets over others was determined. This requires the asset factories to be ordered. If the assets are all created within one factory, their ordering can be handled by the developer within that factory. However, if the assets are defined in independently developed factories, there is a need to mention the order in which the assets need to be created, and consequently the order in which the factories are being called, which can be automatically ordered by the Module Manager. This type of dependency is one of the consequences of aiming at creating an extensible framework, and is the equivalent of inheritance in object programing.

In order to solve this problem, partial ordering of asset factories is used where an attribute (*Predecessors*) allows defining which other factories need to be called before this one. Predecessors that are not included in the current model are ignored, so that maximum flexibility is allowed. For example, if factory F has predecessors A and B, it is possible to run models with any combination of F, A and B. If only A and F are part of the model, then F will automatically be run after A, while B is ignored.

Because of the separation of the *Assets* and the *Behaviours*, this ordering is only necessary on the assets which describe the underlying network structure of the model. Behaviour factories are called after all the asset factories, in any order, since behaviour creations are independent of each other - they communicate only via the assets.

Ordering of the agents' execution

In a "classic" agent-based implementation, agents are often instantiated and scheduled in a central location, e.g. in Repast all the agents are defined in a class implementing *ContextBuilder* (Collier 2014), and in MASON (Luke et al. 2005) in a class extending *SimState*. This means that it is possible to not only create the agents in a given order but also order the agents execution within a time step in relation to other agents. The same capabilities needed to be available with our dynamic composition. As mentioned previously, ordering the behaviours' creation is not important in MODAM, only the ordering of the assets' creation is. However, ordering the execution of the behaviours (i.e. within one step of a simulation) is extremely important, and is necessary to enable deterministic simulation results, which is one of our M&S application requirements for verification purposes.

When talking about scheduling of agents, we do not mean that there exists a central planner that will decide on the actions of the agents but rather on the timing of these actions, which corresponds to the execution of the behaviours. The actions themselves are still undertaken in an autonomous manner by the various behaviours. At the start of the simulation, a global scheduler analyses the dependencies between the behaviours and sorts them into a safe execution order.

Each *BehaviourFactory* creates a set of behaviours, and groups them into named *behaviour groups,* which are used to help order the execution of the behaviours within one timestep. One or many behaviour groups can be defined within a single factory. For example, a BatteryBehaviourFactory could create two types of battery behaviours with different battery control algorithms but that can be executed within the same time step with no specific order. In this case, both kinds of battery behaviours will be assigned the same group. However, if one type of behaviours needs to be executed before the other within the same timestep, an order needs to be specified. For example, if the premise batteries need to be executed first, followed by the grid battery to support the network voltage, then two groups, a PremiseBattery group and a GridBattery group, would be created. This would then allow the ordering to be fully managed by the user who can specify when these groups of behaviours need to be executed. The ordering of the behaviour groups is set through an argument in the command line, followed by the name of the behaviour factory.

An example of behaviour group ordering is given in Figure 6. It shows three behaviour groups, two of which can be run in parallel (*BehaviourGroup A* and *BehaviourGroup B*), that is with no particular order, with the third one requiring its behaviour to be called after both of them. In each of them we have three behaviours: *BehaviourGroup A* and C have their behaviours ordered sequentially, and *BehaviourGroup B* has two behaviours that are ordered sequentially (b_1 and b_2) and one that can be called anytime (b_3). In the example given in Figure 6, the ordering argument in the command line looks like:

$$-order \; = \; (down\,(BehaviourGroup\,A)|up(BehaviourGroup\,B))\;;\; up(BehaviourGroup\,C)$$

where *up* stands for bottom-up, and *down* for top-down ordering; | shows that *BehaviourGroup A* and *BehaviourGroup B* can be ordered in parallel, and the semi-colon (;) is to show sequential ordering.

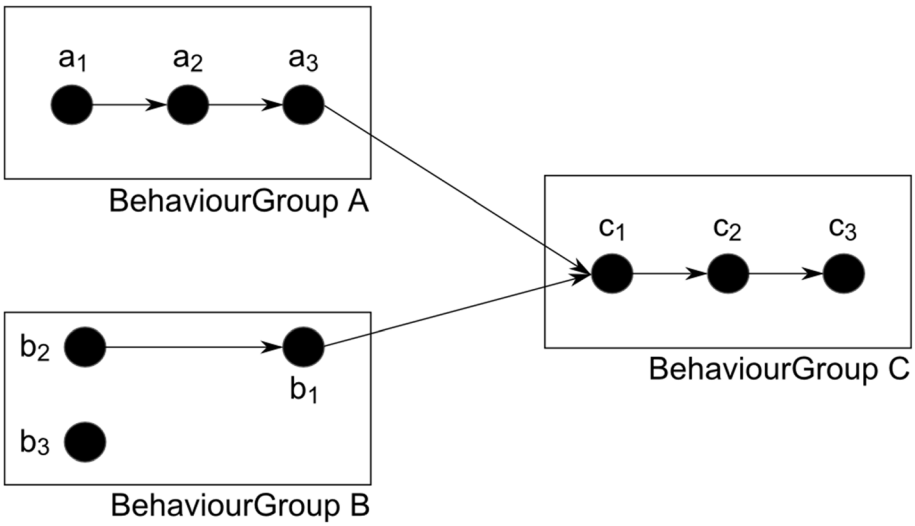

Figure 6 Example of ordering of behaviours within and amongst behaviour groups.

State of the agents at a global scale

Because agents can be developed independently and only come together at runtime to create the agent-based model, keeping track of the state variables can be challenging. Indeed, with a fragmented approach, it is expected that different types of state variables are defined by the developers; however, they still need to be accessed within the *ABMState* to allow tracking the state of the system, and also for other agents to be able to access their value to make their own decisions. To remedy this challenge, we used channels, as explained above, which are effectively globally-named and are typically used as state variables of the simulation. This has the additional advantage that being global variables, they do not necessarily need to be state variables but can also be used for other purposes, such as environmental observation (e.g. local temperature, humidity) or globally observable behaviour attributes (e.g. battery charging strategy).

Explosion of the number of assets and behaviours options

The flexibility in separating Assets and Behaviours can have its downfall. It can quickly become very difficult to know what types of assets and behaviours are currently implemented and which ones are able to come together to form meaningful agents. In addition, knowing what types of parameters or data providers are settable from the command line can also quickly become overwhelming.

One of the principles to follow in this case is that, just because it is possible to break down the system into many simpler components does not mean that it should be done. In some cases, it might be advantageous to keep some groups of assets together, or have many rules within one behaviour especially if it will not be reused elsewhere in the future. This concerns mainly the implementation of the factories, and still implies the separation of assets and behaviours, however many assets or behaviours can be created and put in relation within one factory. An example of this, previously mentioned in this paper, is the asset network factory, which creates many different types of assets (lines, buses, switches, transformers, and premises) and puts them in relation to one another. This implementation was chosen because information about these assets and

their connections were available at once, and contained in files. Also, within the context of our domain application, this represents the current configuration of the distribution network over which transformations will happen. While a factory was defined in this way, it would still have been possible to create each set of assets independently and create the network using partial order.

Despite using this principle, it is still expected that a large number of factories and datasets will be defined and it can be difficult to keep track of which ones are available. This can be supported by good documentation of the software. To facilitate this, we automated the process of documentation, so that as the model is growing, so is the documentation. Documentation is not only useful for the programmer who would like to add a new plugin for example, but also for the user who will not have access to the code, and who will not want to have to go through it. We have used annotations on asset and behaviour factories and on the methods within the classes that are used for input parameters. These annotations are discovered automatically by the documentation generator, and used to generate consistent documentation for the parameters and required data providers of each factory.

Finally, as the model grows, so will the command line. To prevent having too many parameters to define, and also build on previous simulation runs, it is possible to use a configuration file previously saved, to which additional factories, data providers and parameters are specified. This further extends our goal of flexibility in setting up ABM simulations.

Benefits in using a dynamic agent composition

Benefits in having a clear structure of large-scale agent-based models, through the application of the dynamic agent composition, can be identified from the point of view of the software developers for which it was initially designed. Additional benefits can be identified from the perspective of the clients, or users, as well as for agent-based modelling as a scientific approach. These are discussed below.

Benefits from the point of view of the software developers

The dynamic agent composition was adopted to answer shortfalls initially identified when using existing model building approaches and software systems such as Repast and MASON, see Table 1. This approach enabled flexibility and extensibility of both the model definition and its implementation.

Thanks to the distinction between Assets and Behaviours it is easy to change the behaviour of the entities represented in the model. This sometimes needs to happen not only during the model creation but also during the simulation run where they can be added to an existing asset. In ((North and Macal 2007), chapter 7), the authors explain four model growth paths when building agent-based models: the addition of compatible behaviours, contentious behaviours, compatible agents and contentious agents. These four model growth paths are fairly common, and are supported by the dynamic agent composition approach.

Further, the separation into Assets and Behaviours allows gathering certain entities into groups, when relevant, where all assets that are in relation to one another can be defined at once, while different behaviours can be tried separately over each of them without additional development time. This mix-and-match of entities to create agents

can be done at runtime without the need to modify any code; only the command line argument will be altered.

Additionally, having the data separated from the assets and behaviours also makes it easier and faster to extend the model. For example, if a different network is to be modelled, only the asset classes will need to be informed by different data. A new data reader might be implemented if the data format is different; the rest of the code describing the asset and behaviour properties will remain the same. This new data provider will then be called with its associated file at simulation setup, the rest of the command argument remaining the same. This simplifies handling complex options as these are only set in the command line and do not need to be defined in a central class within the code.

Finally, this separation of the agent's aspects into assets, behaviours and data, allows starting an implementation of a model without needing access to all the required information, whether it is data, assets or even a rule that defines the agents' behaviour. This enables an agile implementation (Thomas and Hansmann 2010; Dingsøyr et al. 2010) of the agent-based model. Also, because the dynamic agent composition is implemented in plugins, it is possible for independent authors to contribute to the model in parallel, which can greatly increase implementation time.

Benefits from the point of view of the users

MODAM was also developed to answer requirements from the users' perspective, which the dynamic agent composition enabled.

Various scenarios can be created with ease by simply changing parameters values, data defining the underlying structure of the network but also behaviours. Having the behaviours independent of the assets makes it easy to add and remove behaviours and assess their impact on the system by setting multiple simulation runs. Further, independent teams can have different implementations of a behaviour, providing customisation of their model to better answer the needs of their analyses. In both cases, there is no need to get into the code; the user can simply bring the building blocks together, by specifying them in the command line.

It is also possible to have a mix of behaviour methods to inform a specific type of assets. For example, if it is expected that a given proportion of the population will behave in a certain way, and the rest in another when using a given asset, these two behaviour types can be applied to the model with varying levels by simply calling on these two behaviours and setting a proportion parameter in the command line. This allocation can be done evenly over the population, or be influenced by known factors such as demographic or geographic characteristics. This has the advantage to represent more accurately how the system might evolve if these proportions are known. When unknown, sensitivity analyses over these allocation levels can be performed to find out what mix would be best for the system. This might be useful for educators, for example, who are trying to bring behavioural change and need to find out the population size to target. Simply varying behaviour calls and parameters values without coding will enable them to quickly set up scenarios.

Finally, the use of data to populate the agents offers the advantage of accurately representing a system, which is not widely done especially for large-scale ABMs, and can be of great benefit for the user. Indeed, many large-scale ABMs are currently

developed using taxonomies of agents and probability distributions to represent the system. While such an approach can still be taken using MODAM, we are able to use specific data of the domain of study, which offers the additional benefit of greater fidelity to the system represented. For our electricity network, our asset factories constructed assets based on a data file extracted directly from the Ergon Energy database, so had access to the real attributes of each asset, and their connection to one another. This was a requirement of our client, who is interested in knowing as accurately as possible what is likely to happen on their network at specific locations.

Benefits from the point of view of ABM as a scientific approach

Taking the dynamic agent composition approach when developing MODAM also highlighted advantages in terms of agent-based modelling as a scientific approach.

The separation of an agent into asset and behaviours creates a natural simulation space over which ABMs of networked structures are represented. Indeed, the assets and their connections form a complex network representing the overall structure, which becomes the space over which the behaviours interact with one another. In many simulations such as the Schellings' model (Schelling 1971), a grid is defined as a 2D matrix, over which the behaviours will evolve and get information to make their decision. Here, the idea is similar where the environment is represented by a scale-free network made of the assets which are publicly available within the model and allow every entity to know their relationship to one another. The behaviours are not bound by structure directly but access the underlying network through their asset. The behaviours only contain private data on which they make their decisions. The assets hold the state variables of the ABM simulation which are modified by the behaviours as they make their decisions. While the assets' connections form a network of their own, references to their geo-location (longitude and latitude) are maintained, allowing displaying the network using spatial information software.

MODAM allows replicability of simulations results, thanks to its deterministic capability through each random seed. While independence of the execution of the agents is still ensured through randomisation of their decisions output, having a deterministic order of their execution allows replicability of the experiment and reproducibility of the results.

Finally, MODAM facilitates model comparisons and validation of behavioural sub models. Indeed, data can be used to set the parameters of sub models, that some behaviours use to inform their decisions. As an example, our implementation of solar PV behaviours can be informed by historical solar PV output data or weather data, see Figure 4. The weather data is used to populate a weather-driven model of expected electrical output of solar PV subject to weather with the passage of clouds, while the historical solar PV data simply gives the electrical output of specific solar PVs recorded over a period of time. Because these two approaches require different data input, two data readers were written to be used by each method. When the user chooses their preferred behaviour method, the required data provider will then be called upon. This has the advantage of extending the model, but also offers model comparison and validation of the behavioural sub models. Indeed, the weather-driven solar PV output model could be validated by comparing its output with the actual observations of solar PV output. The dynamic agent composition therefore has the additional benefit that validation

of models can be done easily by setting two simulations and simply changing the datasets and data providers, and comparing their outputs.

Finally, MODAM was developed in Java using open-source frameworks (Eclipse, EMF) and standards (OSGi), ensuring that it is soundly constructed, but also enabling it to be reproduced or extended by interested users.

Related work

Agent-based modelling, a bottom-up modelling technique, describes autonomous agents and their relationships at a fine level of detail with the view of capturing the dynamics of a complex system (Macal and North 2010; Bonabeau 2002). Many toolkits are available to support the development of agent-based models (Nikolai and Madey 2009; Berryman 2008; Railsback et al. 2006; Najlis et al. 2001; North 2013; Luke et al. 2005) with most containing the following features: agents, a scheduler, an interaction space, random number streams, logging and a user interface (North 2013). In these toolkits, agents are generally made up of a unique identifier, behaviours that can be activated and attributes that can be modified. Both the static information and the behaviours are then held in one place, often defined within a class, because object-oriented programing is well suited for agent-based modelling implementation. While held in one class, however, behaviour implementations might still be the result of the composition of behaviours extending others, as is the case for example for the JADE architecture (Bellifemine et al. 2007), where these are extending behaviour classes (Bellifemine et al. 2010). However, these behaviours still are to be added within an agent's implementation, whose architecture is partly hidden, and which requires coding to define the agent. The dynamic agent composition presented in this paper distinguishes itself from these ones as behaviours can be combined without the need to access the agent's code. The behaviours are combined with an asset to form an agent, which can be done by a non-programmer, through a command line, and this composition of the agent happens at runtime.

Our dynamic approach of composing the agents rather follows some of the principles described in MALEVA (Briot et al. 2006) where composability of behaviours is described. While similarities exist between our conceptual frameworks, with behaviours being composed to form a more complex one, they differ in some aspects. MALEVA uses connectors and has output interfaces so that the output of one behaviour is the input of another one to form a chain of complex behaviours. This is not our chosen approach as we are rather more interested in alternative behaviours. While we can use many behaviours to compose one, we do not have this element of precedence of the way the behaviour is executed. Further, our composition concentrates on bringing asset characteristics and behaviours together to form an agent rather than bringing behaviours together, whose need arose from the requirements of growing models, especially when dealing with large-scale ABMs. Because our behaviours communicate via the assets, we also have more flexibility to mix-and-match of the different combinations than MALEVA. Our approach is rather closer to the one specified in (Bae et al. 2012) where a hierarchy of models composed of an action model, an agent model and a multi-agents model has been defined. This is done so that agent-based models can be built incrementally and in a flexible manner, which goal is the same as ours. The authors have presented a formal specification using DEVS (Discrete Event System Specification)

formalism (Zeigler 1976), which they have applied to show that two models could be formalised independently and brought together. It is unclear however that more than two models could be brought together easily, and how large the agent-based models can be. Also, to our knowledge, there is currently no implementation platform to support this specification.

Growing large-scale ABMs is not well documented in the literature, which mainly concentrates on the speed of execution of simulations rather than the modelling needs. However, Parry, in (Parry 2012) differentiates two problems when increasing the scale of a multi-agent system, which includes the computational resources but also the increase in difficulty with a growing agent-based model. Despite this distinction, most of the paper however focusses on how to deal with large-scale simulations which suggests optimising existing code, considering simple solutions such as upgrade of the hardware or evaluating the suitability of the chosen scaling solution on a simplified version of the model. Other approaches to dealing with large-scale simulation requirements is the use of alternative computational techniques, such as considering distributed or parallel implementations of the agent-based model. To this end, simulation toolkits now offer parallel and distributed implementations of their initial implementations as is the case for Repast and MASON for example (Cordasco et al. 2013; Collier 2013), amongst many others. These allow running larger simulations while still getting reasonable execution time, as described in (Parker 2007), where an epidemic simulation runs up to 6 billion agents using a distributed simulation. While the focus of this paper is not on large-scale simulations but rather large-scale model, such challenge is also at the heart of our problem. For this, we have implemented a parallel implementation of our ABM scheduler, as described in (Boulaire et al. 2013b), however, this is not the subject of this paper.

Conclusion

This paper has defined *dynamic agent composition*, a novel approach to build large-scale ABMs. This approach had for goal to extend, with ease, an agent-based model with an underlying networked structure. Also, it aimed at having it flexible so that many scenarios could be created using large corporate databases, without the need for a programmer to build the simulations. By breaking down the model into components containing the data, the assets and the behaviour descriptions, and providing a mechanism to bring them together at runtime to compose the agent-based model, this was achieved.

This approach has many advantages over the way agent-based models are traditionally built for the user as well as the developer. Developers can extend the model without the need to access or modify previously written code; they can develop groups of assets and behaviours independently and add them to those already defined to extend the model. The model can then evolve as new agents need to be modelled, which facilitates the models to be used and extended after previously-defined goals are modified. Users can mix-and-match already implemented asset and behaviour components to form large-scales ABMs. This allows them to quickly setup simulations and easily compare various scenarios without the need to program.

Further, using data extracted from corporate databases to populate the ABM enables to represent accurately the physical infrastructure over which the agents evolve. This

aspect of our approach contributes to the application of ABMs to the electricity domain as most ABMs use taxonomies or probability distribution to represent the network under study.

Future work includes continued expansion of the model to include more asset types as well as behaviours of assets' usage, such as small-scale generators other than solar PVs, and electric vehicles with different charging methods. While currently applied to the electricity distribution grid only, it is expected this approach can be used more broadly and be of benefit to other applications, especially those that have a networked structure, such as water or gas networks.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
FB: has been lead author in drafting the manuscript, has designed and implemented the dynamic agent composition method, has conducted experiments, and has revised the paper. MU: has made key contributions to the paper conception, has aided in the design of the dynamic agent composition method and co-implemented it. RD: has reviewed and revised the paper for important intellectual content. All authors read and approved the final manuscript.

Acknowledgement
The authors gratefully acknowledge the funding through the NIRAP grant, which is making this research possible, the contributions of diverse partners on this project, and especially Ergon Energy for supplying us with data of their network and guidance in terms of tools requirements.

Author details
[1]Queensland University of Technology, Gardens Point – P Block level 8, Brisbane Qld 4000, Australia. [2]University of the Sunshine Coast, Sippy Downs QLD 4556, Australia. [3]Work undertaken at QUT, now at University of the Sunshine Coast, Sippy Downs QLD 4556, Australia.

Received: 27 November 2014 Accepted: 30 January 2015
Published online: 15 February 2015

References

Argonne National Laboratory. The Repast Suite. 2014. http://repast.sourceforge.net/. Accessed 11/11/2014 2014.
Bae JW, Lee G, Moon I-C. Formal specification supporting incremental and flexible agent-based modeling. 2012 Winter Simulation Conference; 2012. Berlin, Germany: IEEE; 2012.
Batten DF, Grozev G, NEMSIM. Finding ways to reduce greenhouse Gas emissions using multi-agent electricity modelling. Complex science for a complex world: exploring human ecosystems with agents. vol Book. Canberra: ANU E Press; 2006. p. 227–52.
Bellifemine FL, Caire G, Greenwood D. Developing multi-agent systems with JADE. Wiley series in agent technology, vol book, whole. Hoboken, New Jersey, USA: Wiley-Blackwell; 2007.
Bellifemine F, Caire G, Trucco T, Rimassa G. JADE Programmer's Guide 2010. Free Software Foundation: Boston, MA, USA
Berryman M. Review of Software Platforms for Agent Based Models. In: Department of Defense, editor. Edinburgh, South Australia, Australia: Defence Science Technology Organisation; 2008.
Bonabeau E. Agent-based modeling: methods and techniques for simulating human systems. Proc Natl Acad Sci U S A. 2002;99(Suppl 3(3)):7280–7. doi:10.1073/pnas.082080899.
Boulaire F, Utting M, Drogemuller R, Abeygunawardana A, Ledwich G, Bell J. Planning for the impact of distributed solar energy on the grid. Solar 2012 Conference; 6–7 December 2012. Melbourne: Swinburne University of Technology; 2012a.
Boulaire F, Utting M, Drogemuller R, Ledwich G, Ziari I. A hybrid simulation framework to assess the impact of renewable generators on a distribution network. 2012 Winter Simulation Conference; 9–12 December 2012. Berlin, Germany: IEEE; 2012b.
Boulaire F, Utting M, Drogemuller R, editors. MODAM: A MODular Agent-based Modelling Framework. 2nd International Workshop on Software Engineering Challenges for the Smart Grid as part of 35th International Conference on Software Engineering (ICSE 2013); 2013a 18–26 May 2013; San Fransisco, CA, USA: IEEE Press
Boulaire F, Utting M, Drogemuller R. Parallel ABM for electricity distribution grids: a case study. 1st Workshop on Parallel and Distributed Agent-Based Simulations, Euro-Par 2013; 26/08/2013. Aachen, Germany: Lecture Notes in Computer Science; 2013b.
Briot J-P, Meurisse T. A Component-based Model of Agent Behaviors for Multi-Agent-Based Simulations. Proceedings of the 7th International Workshop on Multi-Agent-Based Simulation (MABS'06), 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'2006). New York, NY, USA: Association for Computing Machinery (ACM); 2006. p. 183–90.
Cai C, Jahangiri P, Thomas AG, Zhao H, Aliprantis DC, Tesfatsion L. Agent-based simulation of distribution systems with high penetration of photovoltaic generation 2011 24-29/07/2011. San Diego, CA: Power and Energy Society General Meeting, IEEE; 2011.
Castiglione F. Agent based modeling. Scholarpedia. 2006;1(10):1562. doi:10.4249/scholarpedia.1562.
Collier, N. Interface ContextBuilder<T>. 2014 http://repast.sourceforge.net/docs/api/repast_simphony/index.html.

Collier, N. Repast HPC Manual. 2013. http://repast.sourceforge.net/docs/RepastHPCManual.pdf.
Cordasco G, Chiara RD, Raia F, Scarano V, Spagnuolo C, Vicidomini L. Designing computational steering facilities for distributed agent based simulations. Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation. Montreal, Quebec, Canada. 2486147: ACM; 2013. p. 385–90.
del Valle Y, Venayagamoorthy GK, Mohagheghi S, Hernandez JC, Harley RG. Particle swarm optimization: basic concepts, variants and applications in power systems. IEEE Trans Evol Comput. 2008;12(2):171–95. doi:10.1109/tevc.2007.896686.
Dingsøyr T, Dybå T, Moe N. Agile Software Development: An Introduction and Overview. In: Dingsøyr T, Dybå T, Moe NB, editors. Agile Software Development. Berlin, Germany: Springer Berlin Heidelberg; 2010. p. 1–13.
Ergon Energy. Corporate profile. 2013. https://www.ergon.com.au/about-us/who-we-are/our-company/corporate-profile. Accessed 02/06/2013 2013.
Gamma E. Design patterns: elements of reusable object-oriented software. vol Book, Whole. Boston: Addison-Wesley; 2009.
Hamill L. Agent-based modelling: The next 15 years. JASSS. 2010;13(4):7.
Institute for Energy and Transport. Agent Based Modelling for Smart Grids. European Comission. 2014. http://ses.jrc.ec.europa.eu/agent-based-modelling-smart-grids. Accessed 20/07/2014 2014.
Klügl F, Bazzan ALC. Agent-based modeling and simulation. AI Mag. 2012;33(3):29–40.
Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G. MASON: a multi-agent simulation environment. Simulation. 2005;82(7):517–27.
Macal CM, North MJ. Tutorial on agent-based modelling and simulation. J Simul. 2010;4:151–62.
Macal CM, North MJ, editors. Tutorial On Agent-Based Modeling And Simulation Part 2: How To Model With Agents. Winter Simulation Conference; 2006 December 3–6, 2006; Monterey, California, USA; SI.l: Omni press
Morton A. A fast 'do-it-yourself' load flow algorithm for power systems with sparse topology. AUPEC 2003 Australasian Universities Power Engineering Conference. Christchurch, New Zealand: University of Canterbury, NZ; 2003.
Najlis R, Janssen MA, Parker DC, editors. Software Tools and Communication Issues. Proceedings of a Special Workshop on Land-Use/Land-Cover Change; 2001 04-07/10/2001; Center for Spatially Integrated Social Science, University of California at Santa Barbara: Irvine, California 2002.
Nikolai C, Madey G. Tools of the trade: a survey of various agent based modeling platforms. J Artif Soc Soc Simul. 2009;12(2):2.
North MJ. A theoretical formalism for analyzing agent-based models. Complex Adaptive Systems Modeling. 2013;2(1):3. doi:10.1186/2194-3206-2-3.
North MJ, Macal CM. Managing Business Complexity. vol Book, Whole. New York, NY, USA: Oxford University Press; 2007.
North M, Conzelmann G, Koritarov V, Macal C, Thimmapuram P, Veselka T. E-laboratories : agent-based modeling of electricity markets. American Power Conference; 15-17/04/2002. Chicago, IL (US): Argonne National Lab., IL (US); 2002.
Parker J. A flexible, large-scale, distributed agent based epidemic model. 2007 Winter Simulation Conference; 2007. Washington, DC, USA: IEEE Press; 2007.
Parry HR. Agent based modeling, large scale simulations. New York, NY: Springer New York; 2012. p. 76–87.
Railsback SF, Lytinen SL, Jackson SK. Agent-based simulation platforms: review and development recommendations. Simulation. 2006;82(9):609–23.
Schelling TC. Dynamic models of segregation. J Math Sociol. 1971;1:143–86.
Steinberg D, Budinsky F, Paternostro M, Merks E. EMF: Eclipse Modeling Framework Eclipse Series. Boston, MA, USA: Addison-Wesley Professional; 2008.
The Eclipse Foundation. About the Eclipse Foundation. The Eclipse Foundation. 2012. http://www.eclipse.org/org/. Accessed 27/02/2012 2012.
Thomas S, Hansmann U. Overview of Agile Software Development. Agile Software Development: Best Practices for Large Software Development Projects. vol Book, Whole. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 35–9.
Weidlich A. Engineering interrelated electricity markets: an agent-based computational approach. vol Book, Whole. Heidelberg: Springer; 2008.
Zeigler BP. Theory of modelling and simulation. vol Book, Whole. New York, N.Y: Wiley; 1976.