Specification and Validation of the MODAM Module Manager

Mark Utting[1] and Fanny Boulaire[2]

[1] University of the Sunshine Coast, Australia, `utting@usc.edu.au`
and The University of Waikato, New Zealand, `marku@waikato.ac.nz`
[2] Queensland University of Technology, Australia, `Fanny.Boulaire@qut.edu.au`

Abstract. Electricity distribution networks are large complex systems that are continuously evolving. Agent-based models are a useful way of exploring possible future scenarios for these networks. This paper introduces MODAM, a software framework developed to support building large-scale agent-based models for electricity distribution network planning. This paper describes how models can be assembled in an automated manner at runtime, even though an agent may be composed of aspects that come together from separate components. The Module Manager, which weaves the components together in an automated manner, is described in this paper using formal specifications written in Z, and the specification is validated using the ZLive animation tool.

Keywords: Agent-based modelling, automation, modularity, composition, networked structures, Z, ZLive, specification validation, animation

This paper describes how the MODAM (MODular Agent-based Model) framework can automate the construction of a complex agent-based model (ABM) from a set of user-chosen components. The components are designed and implemented by programmers, but the selection, instantiation, and wiring together of those components can be done by modellers who are not programmers, thanks to the automated model-building features of MODAM. We specify the key aspects of this model-building process in Z, and we use the ZLive animation tool to validate that our specification has the desired properties. We also discuss several features of the ZLive tool that make it more convenient to use for this kind of experimental exploration of the properties of a Z specification.

Sections 1 and 2 give a brief introduction to future electricity grids and the MODAM framework. Sections 3 to 5 describe the three aspects that support the automated building of large-scale agent-based models using dynamic agent composition: the modularity technology, how components can be composed to form a model, and finally how this composition process can be automated using a Module Manager. We formalise key aspects of these stages, using Z. In Section 6, we validate some of important properties of these specifications using ZLive, and in Section 7 we reflect upon the process of using ZLive for validating specifications, and discuss how ZLive's 'why' command is useful for debugging difficult specifications.

1 Future Electricity Networks: Motivation for MODAM

Electricity distribution networks are undergoing rapid changes with the introduction of new technologies, policies and demand management options. For example, a study done by the Future Grid Forum predicts that batteries in combination with energy efficiency measures, gas generation and solar panels could lead a third of the customers to leave the grid in Australia by 2050 [4].

To better plan the future grid, planning tools used by decision-makers need to take into account the new technologies and new approaches that may impact the future grid. We have developed a simulation environment, called MODAM (MODular Agent-based Model), to assess the impact of different trajectories of consumption at varying locations of the network over many years. It supports understanding the changes in load at every node in the network with the introduction of new technologies (e.g. solar panels, batteries), new policies (e.g. time-of-use tariffs) or/and demand management. It was developed using agent-based modelling [5], which was chosen for its capacity to represent at a fine level of detail the behaviours and interactions of the network's entities that have a spatial and temporal component, as well as the behaviours of the different consumers impacting it. Agent-based modelling has been successfully used for different applications in the electricity sector [7, 10, 3], for properties such as those mentioned above, and is particularly suited to our application domain. Ergon Energy, a Queensland electricity distribution company, commissioned this project to perform simulations of the future of their grid. To answer their needs, the following software requirements were drawn:

- A large-scale agent-based model is to represent the Ergon's distribution network, based on large data sets coming from corporate databases;
- From there, trajectories of consumption are to be simulated using different assumptions and knowledge about the type of technology that will likely be taken up, their location and the way it is going to be used;
- The model can grow and change easily over the time of the project and beyond it, bringing extensibility and flexibility in the model definition;
- A vast range of scenarios needs to be tried easily, and this can be done on a daily basis by an engineer, without the need to code.

2 Automated Model Construction in MODAM

MODAM models are constructed at runtime from a collection of user-specified modules. Figure 1 gives a schematic representation of the different building blocks, also called components or modules that are currently available in MODAM. For example, if we are building a simulation of a distribution network containing premises with solar panels and batteries, we would use the modules of Network Assets, PV assets and Battery Assets. The choice of behaviour modules for these assets might vary, depending on the goals of the simulation. Data modules would also be chosen to specify the characteristics of the assets or the behaviours, and this data could come in various formats, requiring different reader modules.

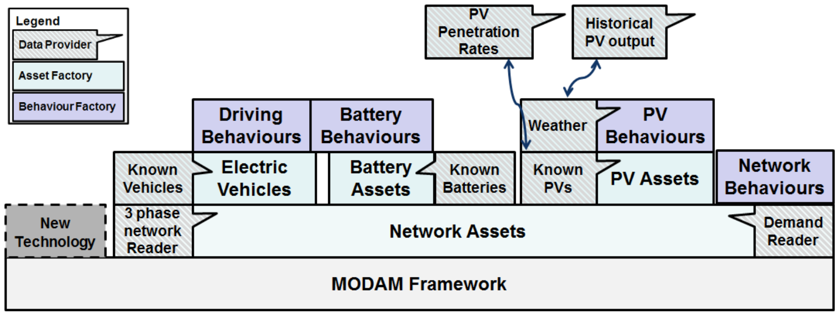


Fig. 1. The MODAM framework with some modules for modelling electricity networks.

To support building flexible and extensible large-scale agent-based models with many different behaviours, we developed a *dynamic agent composition* technique for building agent-based models [2]. It consists of breaking down an agent into an *asset* part and a set of *behaviours*, where these aspects are defined separately in the model definition and its implementation.

$$Agent = Asset + Behaviours$$

These two aspects then come together at runtime only, to form what is traditionally called an agent (made of attributes and behaviours [8]). This method facilitates building an agent-based model incrementally, as well as offers many options for agents to be defined using alternative or combining behaviours. With this dynamic composition approach, it would be possible to take a manual approach to bringing the agents together at runtime into a coherent agent-based model. However, this would require a modeller to code, which does not answer our last requirement of code-free set up of simulations. Consequently, we have automated this model-assembly process, which is the subject of this paper.

A *Module Manager* weaves together modules that hold information about the agents, to compose a model at runtime, in an automated manner. Three aspects support this automated process – these are the focus of this paper and will be discussed in the following sections:

– Technical aspects to support modularity – the definition of modules, and the technology to compose them at runtime;
– Collaboration between the user and MODAM to specify which modules and data should be composed;
– An automation aspect, where the module manager links the assets, behaviours and data to create an ABM, on which simulations are run.

3 Specifying a MODAM module

A key criteria when building our ABMS application was for it to be a flexible and extensible model environment. We chose OSGi (formerly Open Services Gateway

Initiative) as the technology to support modularity, as it allows modules to be installed, started, stopped or uninstalled at runtime, and it is used extensively in Eclipse. Each MODAM module is implemented as an Eclipse plugin, which allows MODAM to easily assemble models from many plugins from a variety of sources.

Using the Eclipse plugin architecture, the MODAM framework defines three *extension points*: **AssetFactory**, **BehaviourFactory** and **DataProvider**. Each plugin that defines an extension of these extension points specifies a Java class name that will be used as the factory to create the assets, or behaviours, or data outputs. In addition, each extension of **DataProvider** has a unique identifier and a path parameter to indicate the default input file to be used; while extensions of **AssetFactory** and **BehaviourFactory** have a set of *'consumes'* identifiers that specify the kinds of input data that they need.

We specify the extension points and factories in the Z specification language [1], using the Z section facilities to break the specification into several parts. The first Z section defines the basic types of objects used by MODAM.

$$\textbf{section}\ \mathit{ModamTypes}\ \textbf{parents}\ \mathit{standard\_toolkit}$$

First we declare the basic types used by the ModuleManager of MODAM.

$$\begin{array}{l}
[\mathit{ClassName}, \mathit{ExtId}, \mathit{DataId}, \mathit{Path}, \mathit{Contributor}] \\
\mathit{OPTIONAL} ::= \mathit{YES} \mid \mathit{NO}
\end{array}$$

ClassName is the set of all possible Java class names, *ExtId* is the set all possible Eclipse extension point identifiers (e.g., `au.edu.qut.modam.assetfactory`), *DataId* is the set of all possible Data identifiers (eg. `NetworkData`) – these are used to link providers and consumers together via named ports, *Path* is the set of all filesystem paths to input data files, and a *Contributor* is a unique identifier for a module. We define an extension point ID for the three key MODAM extension points, plus several constants to represent unknown values, and the **dataProvidedError** to model the exception that is returned when suitable data providers cannot be found.

$$\begin{array}{|l}
\mathit{modamAssetFactory}, \mathit{modamBehavFactory}, \mathit{modamDataProvider} : \mathit{ExtId} \\
\mathit{noDataId} : \mathit{DataId} \\
\mathit{noPath} : \mathit{Path} \\
\mathit{nullClass}, \mathit{dataProviderError} : \mathit{ClassName} \\
\hline
\mathit{modamAssetFactory} \neq \mathit{modamBehavFactory} \\
\mathit{modamAssetFactory} \neq \mathit{modamDataProvider} \\
\mathit{modamBehavFactory} \neq \mathit{modamDataProvider} \\
\mathit{dataProviderError} \neq \mathit{nullClass}
\end{array}$$

We actually have two versions of the above Z section: the one just shown, which uses infinite *given sets* for each of the basic MODAM types, such as class names; and an animation-friendly variant called *ModamTypeAnimate*, which uses small integer ranges for each of these basic types, to make animation of the specification easier. We typically develop and type-check the Z specification using the infinite data type, and switch to the finite one (simply by changing the name of the parent in the following section command) when we want to animate operations, or run unit tests.

section *ModuleManager* **parents** *ModamTypes*

This Z section describes the ModuleManager of MODAM. A key concept is a *SetterGetter*, which represents a property of an object whose value can be read and set via getter/setter methods. It has a name, an argument type, a current value (for Modam, this is always a reference to a Java object that is an instance of the *value* class, which must be a subtype of the *argType* class), and a flag which specifies whether the property is optional or compulsory.

$$\begin{array}{|l}
\textit{SetterGetter} \\ \hline
\textit{name} : \textit{DataId} \\
\textit{argType}, \textit{value} : \textit{ClassName} \\
\textit{optional} : \textit{OPTIONAL} \\ \hline
\end{array}$$

We define the Eclipse concept of *extension point*, which contains a class name plus several properties. We call these extension classes *Factory* classes, because in MODAM they are used to create assets and agents and data provider objects. We define three extension points, for assets, behaviours, and data providers respectively. We use a separate ID for each extension point, and we define various special subclass characteristics for each kind of extension point. For example, asset factories and behaviour factories may consume data providers but they do not produce data, whereas data providers produce data but do not not consume other data providers. Asset factories are the only kinds of factories whose execution has to be ordered, so we set $prior = \varnothing$ for the other factories.

$$\begin{array}{|l}
\textit{Factory} \\ \hline
\textit{extId} : \textit{ExtId} \\
\textit{className} : \textit{ClassName} \\
\textit{consumes} : \mathbb{P}\, \textit{DataId} \\
\textit{produces} : \textit{DataId} \\
\textit{path} : \textit{Path} \\
\textit{prior} : \mathbb{P}\, \textit{ClassName} \\
\textit{methods} : \mathbb{P}\, \textit{SetterGetter} \\ \hline
\end{array}$$

$$\begin{aligned}
\textit{AssetFactory} == {} & [\textit{Factory} \mid \textit{extId} = \textit{modamAssetFactory} \land \\
& \textit{produces} = \textit{noDataId} \land \textit{path} = \textit{noPath}] \\
\textit{BehavFactory} == {} & [\textit{Factory} \mid \textit{extId} = \textit{modamBehavFactory} \land \\
& \textit{produces} = \textit{noDataId} \land \textit{prior} = \varnothing \land \textit{path} = \textit{noPath}] \\
\textit{DataProvider} == {} & [\textit{Factory} \mid \textit{extId} = \textit{modamDataProvider} \land \\
& \textit{consumes} = \varnothing \land \textit{prior} = \varnothing \land \textit{produces} \neq \textit{noDataId}]
\end{aligned}$$

A Module in MODAM is essentially just a contributor ID plus a set of extension classes, which must all have unique names. The *consumes* and *produces* maps are derived by collecting into a single map all the data IDs consumed by all the factories, or produced by all the data providers, respectively. Similar maps are used in the Java implementation of the Module class in MODAM.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Module}} \\
\hline
\mathit{contributor} : \mathit{Contributor} \\
\mathit{classes} : \mathbb{P}\, \mathit{Factory} \\
\mathit{consumes} : \mathit{ClassName} \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{P}\, \mathit{DataId} \\
\mathit{produces} : \mathit{ClassName} \mathrel{+\!\!\!\twoheadrightarrow} \mathit{DataId} \\
\hline
(\forall\, c1, c2 : \mathit{classes} \bullet c1.\mathit{className} = c2.\mathit{className} \Rightarrow c1 = c2) \\
\mathit{consumes} = \{c : \mathit{classes} \bullet (c.\mathit{className}, c.\mathit{consumes})\} \\
\mathit{produces} = \{c : \mathit{classes} \bullet (c.\mathit{className}, c.\mathit{produces})\} \\
\hline
\end{array}
$$

Finally, the MODAM *ModuleManager* simply contains the set of modules that the user has selected to be part of the model, either via the command-line or via a GUI. We define *outputid* to be a mapping that captures key information about *all* the available data providers.

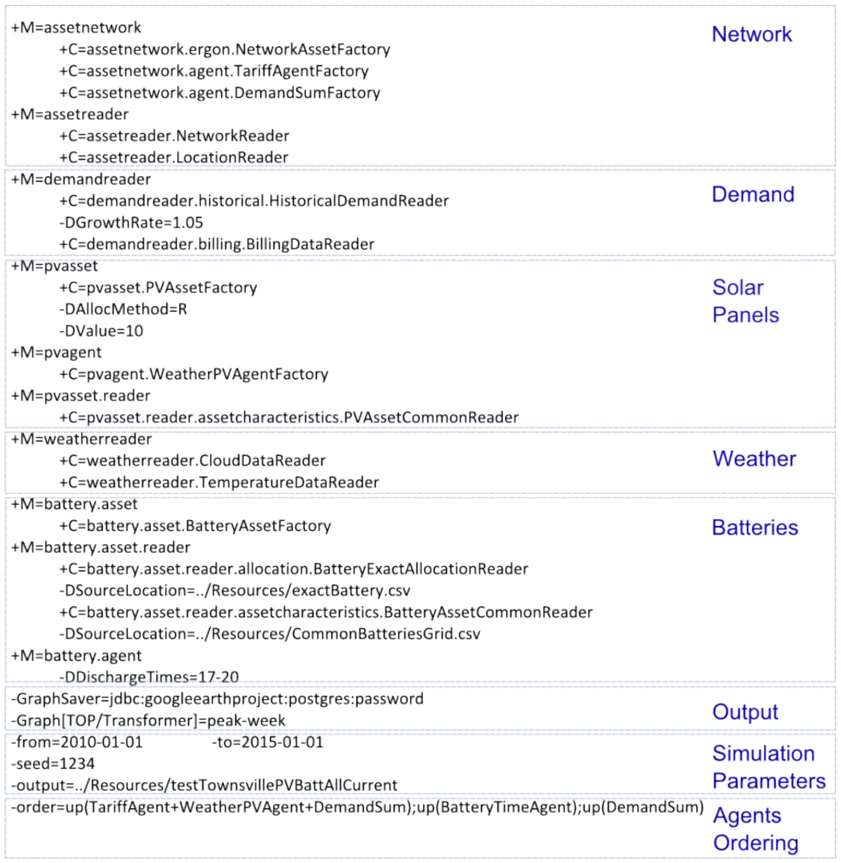
$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ModuleManager}} \\
\hline
\mathit{modules} : \mathbb{P}\, \mathit{Module} \\
\mathit{outputid} : \mathit{ClassName} \mathrel{+\!\!\!\twoheadrightarrow} \mathit{DataId} \\
\hline
\mathit{outputid} = \bigcup \{m : \mathit{modules} \bullet m.\mathit{produces}\} \\
\hline
\end{array}
$$

4 Composing flexible ABMs

This section describes how a network planning engineer can build up a model and run a simulation, without programming. Creating a model is done in a collaborative manner between the user and the MODAM framework. There are currently two main ways of interacting with the ABM simulation software: command line scripts or graphical user interfaces.

4.1 Command line scripts

The command-line scripts are typically used to run a sweep of scenarios on a cluster of computers. This allows many alternative scenarios to be explored at once, or the same scenario run many times with different random seeds so that the people and premises in the simulation have different behaviour each run – then the results of all those runs can be analysed statistically to determine the averages, standard deviations, and Probability of Exceedance (PoE) 50 and PoE 10 levels of demand peaks for each part of the network. There is no need to know about programming, simply knowing which factories have been implemented and

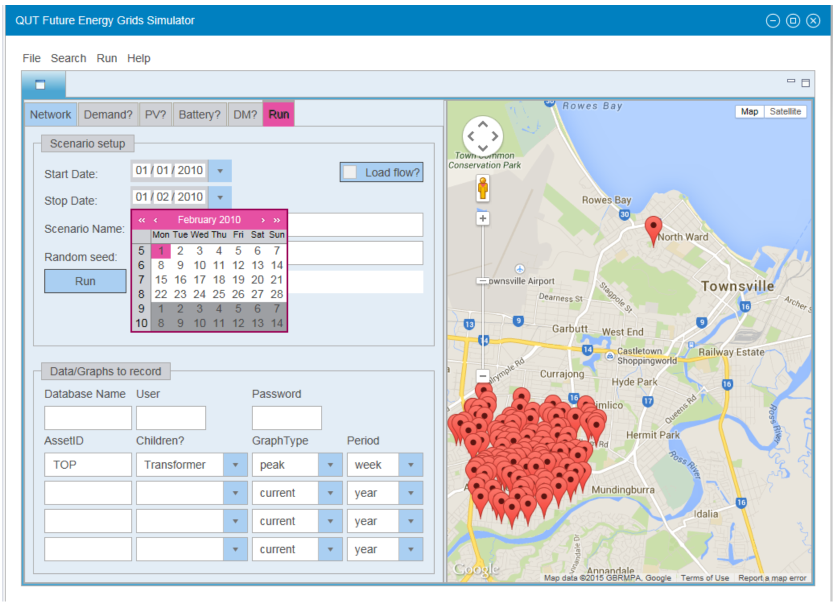


```
+M=assetnetwork
        +C=assetnetwork.ergon.NetworkAssetFactory
        +C=assetnetwork.agent.TariffAgentFactory
        +C=assetnetwork.agent.DemandSumFactory
+M=assetreader
        +C=assetreader.NetworkReader
        +C=assetreader.LocationReader
+M=demandreader
        +C=demandreader.historical.HistoricalDemandReader
        -DGrowthRate=1.05
        +C=demandreader.billing.BillingDataReader
+M=pvasset
        +C=pvasset.PVAssetFactory
        -DAllocMethod=R
        -DValue=10
+M=pvagent
        +C=pvagent.WeatherPVAgentFactory
+M=pvasset.reader
        +C=pvasset.reader.assetcharacteristics.PVAssetCommonReader
+M=weatherreader
        +C=weatherreader.CloudDataReader
        +C=weatherreader.TemperatureDataReader
+M=battery.asset
        +C=battery.asset.BatteryAssetFactory
+M=battery.asset.reader
        +C=battery.asset.reader.allocation.BatteryExactAllocationReader
        -DSourceLocation=../Resources/exactBattery.csv
        +C=battery.asset.reader.assetcharacteristics.BatteryAssetCommonReader
        -DSourceLocation=../Resources/CommonBatteriesGrid.csv
+M=battery.agent
        -DDischargeTimes=17-20
-GraphSaver=jdbc:googleearthproject:postgres:password
-Graph[TOP/Transformer]=peak-week
-from=2010-01-01          -to=2015-01-01
-seed=1234
-output=../Resources/testTownsvillePVBattAllCurrent
-order=up(TariffAgent+WeatherPVAgent+DemandSum);up(BatteryTimeAgent);up(DemandSum)
```

Fig. 2. Example command line for a network with PV panels and batteries.

are available is sufficient. For some factories, additional arguments and datasets can be specified, to populate the ABM and set some simulation parameters.

An example of a command script is given in Figure 2. It describes a simulation of a base electrical network, on top of which solar panels and batteries have been added. The boxes surrounding the commands highlight groups of information, within which modules ("+M"), factory classes ("+C") and parameters ("-D") are added to the simulation. Each class can be customized using `-D`*`name=value`* flags to specify parameter values. The Module Manager finds the modules in the registry and instantiates the specified classes; reflection is used to set the SetterGetter properties of that class to the specified parameter values. The last part of Figure 2 shows parameters for the whole simulation: the start and end dates (`-from` and `-to`), the `-seed` for the random number generator to ensure reproducibility of simulation experiments, and `-output` for the folder that will contain the simulation output. Finally, the `-order` parameter is used to schedule groups of agents in the given order within each timestep.



Start Date: 01 / 01 / 2010

Stop Date: 01 / 02 / 2010

Scenario Name:

Random seed:

AssetID	Children?	GraphType	Period
TOP	Transformer	peak	week
		current	year
		current	year
		current	year

Fig. 3. Web-based user interface for running agent-based network simulations.

4.2 Graphical User Interfaces

We have developed two types of GUIs: a generic one that corresponds closely to the command line (allowing users to select any MODAM modules and classes and set their parameters), and one tailored to the needs of electricity distribution planners. The tailored GUI supports only a limited number of plugins and factories, which can be combined to answer predefined questions. This is more user-friendly for a planner who does not need to know what factories to select and only wants to perform a limited number of analysis types, where the data populating the models and the simulation parameters can still be varied.

An example of the client part of the interface is shown in Figure 3 – this runs in a web browser and communicates with the main ABM program that is running on a server computer. The first tab allows a segment of the network to be selected and viewed geographically on the map. Then the Demand, PV and Battery tabs are used to set up a simulation, and the Run tab (shown) is used to run the simulation on the server for a given simulation time period. The scenario is also saved so it can be used as a basis for command-line batch simulations later.

5 Assembling a model: an automated process

The Module manager, as its name indicates, manages the modules specified by the user in the command scripts or the GUI. To assemble the model and start the simulation, the Module Manager performs the following stages.

1. Find and collect the enabled modules and extension points;
2. Check and warns of missing dependencies;
3. Instantiate one instance of each factory and data provider and process their SetterGetter methods (see definition of SetterGetter below);
4. Call the asset factories, in the correct order using topological sort, to create assets and populate them with data if specified;
5. Use behaviour factories to create behaviours and populate them with data;
6. Call the start method on all the behaviours;
7. Start stepping the simulation.

These seven stages are automated by the module manager, so users do not have to write code to assemble a model. They just have to choose modules, factories and parameters, then the module manager automates the building and running of the model. We shall specify in Z some interesting aspects of this process.

Once the modules specified by the user have been discovered and enabled (Stage 1), the module manager checks for any missing dependencies (Stage 2). The following *GetMissingDependencies* operation finds all the non-optional data requirements that are not satisfied by any of the data providers. If the result *missing*! set is non-empty, this indicates that the current model setup is incomplete, so the model cannot be run.

If there are no missing dependencies, Stage 3 of the module manager instantiates one instance of each selected factory and data provider, sets any integer/string parameters that were specified for those objects on the command line or GUI, and then tries to link the available data providers into the asset and behaviour factories, so that the assets and behaviours can be created and populated by the required data.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{GetMissingDependencies}} \\
\hline
\Xi \mathit{ModuleManager} \\
\mathit{factories?} : \mathbb{P}\, \mathit{Factory} \\
\mathit{missing!} : \mathbb{P}\, \mathit{DataId} \\
\hline
\mathit{missing!} = \{ \mathit{unsatisfied} : \mathit{DataId} \mid \\
\quad (\exists\, m : \mathit{modules};\ f : \mathit{factories?};\ s : \mathit{SetterGetter} \bullet \\
\qquad f.\mathit{className} \in \operatorname{dom} m.\mathit{consumes} \land \\
\qquad \mathit{unsatisfied} \in m.\mathit{consumes}\,(f.\mathit{className}) \land \\
\qquad s \in f.\mathit{methods} \land s.\mathit{optional} = \mathit{NO} \land \\
\qquad s.\mathit{name} = \mathit{unsatisfied} \land \\
\qquad (\lnot\, \exists\, m' : \mathit{modules} \bullet s.\mathit{name} \in \operatorname{ran} m'.\mathit{produces}))\} \\
\hline
\end{array}
$$

One of the important operations in Stage 3 is automatically analyzing each property of a requested factory to see if it can be satisfied by the available data providers. The following *SatisfySetterMethod* specifies how this is done for each property. It transforms the value of a *SetterGetter* method after having matched an *outputid* to its dataProvider from a set of input *dataProviders*. The *matching*! output set contains all the available data providers that can satisfy this property. Then there are four cases:

- the success path corresponds to having exactly one match, which leads to setting the value of the method to that match;
- another path where there is no match found but the matching was optional, leading to an unchanged method;
- an error is thrown if no matches are found and the property is not optional;
- an error is thrown if there are multiple matches, so it is ambiguous which data provider the user intended to be used.

$$
\begin{array}{|l}
\hline
\mathit{SatisfySetterMethod} \\
m, m' : \mathit{SetterGetter} \\
\mathit{dataProviders}? : \mathbb{P}\, \mathit{ClassName} \\
\mathit{outputid} : \mathit{ClassName} \twoheadrightarrow\!\!\!\!\!\!\!+\;\; \mathit{DataId} \\
\mathit{matching}! : \mathbb{P}\, \mathit{ClassName} \\
\mathit{value}! : \mathit{ClassName} \\
\hline
m' = \langle\!|\, \mathit{name} == m.\mathit{name}, \mathit{argType} == m.\mathit{argType}, \\
\mathit{optional} == m.\mathit{optional}, \mathit{value} == \mathit{value}! \,|\!\rangle \\
\mathit{matching}! = \{ d : \mathit{dataProviders}? \mid \mathit{outputid}\ d = m.\mathit{name} \} \\
\#\, \mathit{matching}! = 1 \Rightarrow \mathit{value}! \in \mathit{matching}! \\
\#\, \mathit{matching}! > 1 \Rightarrow \mathit{value}! = \mathit{dataProviderError} \\
\#\, \mathit{matching}! = 0 \land m.\mathit{optional} = \mathit{YES} \Rightarrow \mathit{value}! = m.\mathit{value} \\
\#\, \mathit{matching}! = 0 \land m.\mathit{optional} = \mathit{NO} \Rightarrow \mathit{value}! = \mathit{dataProviderError} \\
\hline
\end{array}
$$

After this Stage 3 has completed, the asset and behaviour factories have been connected to appropriate data providers, so the desired assets and behaviours can now be created. Stage 4 executes each asset factory so that it can create assets (typically in a data-driven fashion) and add them to the ABMState, which is the central class for the simulation. Since the creation of some kinds of assets (e.g. PV systems) may depend on other kinds of assets already existing (e.g. houses to install PV systems on), each asset factory can specify which other asset factories it depends upon, and the module manager sorts these dependencies to ensure that asset factories are run in a correct order.

Once the network of assets is built, Stage 5 executes the behaviour factories to add behaviour objects to the assets (in MODAM an agent is an asset plus zero or more behaviour objects). Stage 6 executes the `start()` method of all the behaviours to initialise the agents, then in Stage 7 the ABMState object steps through the simulation by using its scheduler to execute all the `step()` methods of all the agent behaviours, in an appropriate order. This order is given by the `-order` command line parameter, or is constructed automatically by the GUI.

6 Validation

Experience shows that the first draft of a formal specification such as the above is rarely correct as written. To validate that the specification says what we want, we use the ZLive animation tool to test some typical scenarios. ZLive is the Z animation tool that is part of the CZT tools suite.[3] It evaluates expressions, schemas, and predicates written in Z, using the following steps:

1. Use CZT transformation rules to unfold schema operators and most other Z operators into a core Z subset based on set comprehension expressions;
2. Perform static analysis (abstract interpretation to determine integer ranges and set sizes) to reduce the search space from infinite to finite, and estimate set sizes, where possible;
3. Sort the predicates within each set comprehension, using a greedy *smallest-first* algorithm, so that small sets are iterated through first, and filter predicates are evaluated as early as possible;
4. Lazily enumerate the solutions to each set comprehension, on demand.

For example, if we have defined a *Factors* schema as:

$$Factors == [num, fact : \mathbb{N} \mid fact \in \operatorname{dom}\{a, b : 0..num \mid a * b = num\}]$$

the ZLive command *do Factors* $\wedge$ $[num == 10]$ will return the first solution:

$$1 : \langle\!| num == 10, fact == 1 |\!\rangle$$

then successive '`next`' commands will step through the remaining solutions (*fact* equals 2, 5, and 10), and finally ZLive will report '`no more solutions`'.

Our methodology for validating the specification is similiar to a typical *unit testing* approach. We take some simple real-world use cases of the MODAM module manager and translate their example input modules into Z, then we use those modules as inputs to our Z specification and animate the specification using ZLive to produce some outputs. Finally, we check that those outputs agree with our informal requirements for the module manager. As we shall see, we can also use ZLive to check important properties of our example modules, such as being uniquely defined – this is similar to type checking and determinism checking in some programming languages.

So we define in Z an example asset factory for photovoltaic solar panels, with a compulsory property called *pvCharacteristics* that takes any data provider that is a subclass of *pvCharacteristicsReader* (this provides a list of typical PV systems), and an optional getter-setter called *exactAllocation* that takes a data provider of type *pvExactAllocationReader* (this is used when precise information is available about which houses have PV systems, and the details of those systems).

$$\begin{array}{|l}
\multicolumn{1}{l}{pvCharacteristics} \\
SetterGetter \\
\hline
name = dataIdPvCharacteristics \\
argType = pvCharacteristicsReader \\
value = nullClass \\
optional = NO \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{exactAllocation} \\
SetterGetter \\
\hline
name = dataIdPvExactAllocation \\
argType = pvExactAllocationReader \\
value = nullClass \\
optional = YES \\
\hline
\end{array}$$

[3] See `http://czt.sourceforge.net/zlive` for information about ZLive, and `http://czt.sourceforge.net` for information about the Community Z Tools (CZT) project.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PvAssetFactory}} \\ \hline
\mathit{AssetFactory} \\ \hline
\mathit{className} = \mathit{pvAssetFactory} \\
\mathit{consumes} = \{\} \\
\mathit{prior} = \{\} \\
\mathit{methods} = \mathit{pvCharacteristics} \cup \mathit{exactAllocation} \\ \hline
\end{array}
$$

We then check that these two getter-setters and the factory are correctly and unambiguously defined (we have not defined any contradictory properties, or left any properties unspecified), by defining theorems like the following ($\vdash ? \mathit{predicate}$ is the ISO Standard Z syntax for the conjecture that *predicate* follows from the Z specification [1]):

theorem pvCharacteristicsIsValid
$\vdash ? \ \# \mathit{pvCharacteristics} = 1$

The first time we asked ZLive to evaluate this conjecture, it was false! To investigate why, we asked ZLive to generate members of the *pvCharacteristics* schema one-by-one (‘`do pvCharacteristics`’), and quickly discovered that there were *no more solutions*, so $\# \mathit{pvCharacteristics}$ is actually zero! This was because we had specified the wrong kind of value for `argType`. This is a minor ‘typing’ error that is picked up by the Z typechecker when we use separate given types for each kind of value, or is picked up by the above conjectures when we use just integers for animation purposes. Reflecting on these failures led us to revise our model so that the *value* and *argType* are now both Java class names, representing the actual and expected types of parameter, respectively.

A similar failure happened the first time we tried to prove that our *PvAssetFactory* was well-formed, via the following conjecture. This conjecture was initially false, because we had accidently put class names into the *consumes* set, rather than data IDs. We shall discuss this example further in the next section.

theorem PvAssetReaderFactoryIsValid
$\vdash ? \ \# \mathit{PvAssetFactory} = 1$

Now we test each of the four cases of the *SatisfySetterMethod*.

1. First, we test the success path, where there is exactly one match.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{testUniqMatch}} \\ \hline
\mathit{SatisfySetterMethod} \\ \hline
m \in \mathit{pvCharacteristics} \\
\mathit{dataProviders}? = \{\mathit{pvCharacteristicsReader}, \mathit{pvExactAllocationReader}\} \\
\mathit{outputid} = \{\mathit{pvCharacteristicsReader} \mapsto \mathit{dataIdPvCharacteristics}, \\
\qquad\qquad\qquad \mathit{pvExactAllocationReader} \mapsto \mathit{dataIdPvExactAllocation}\} \\ \hline
\end{array}
$$

In this case, ZLive updates $m'.value$ to the unique matching data provider class *pvCharacteristicsReader* (whose ID is 14), and also returns that class in $value!$. ZLive returns the following in LaTeX Z syntax (we have manually highlighted the generated return values in bold):

$$\begin{array}{l}
1 : \langle\!| m == \langle\!| name == 35, argType == 14, value == 9, optional == 0 |\!\rangle,\\
m' == \langle\!| name == 35, argType == 14, \mathbf{value} == \mathbf{14}, optional == 0 |\!\rangle,\\
dataProviders? == \{14, 17\}, outputid == \{(14, 35), (17, 36)\},\\
matching! == \{14\}, \mathbf{value!} == \mathbf{14} |\!\rangle
\end{array}$$

2. When there are multiple matching data providers, ZLive reports a unique solution, with the $value!$ output set to *dataProviderError* (8):

$$\begin{array}{|l}
\multicolumn{1}{l}{testSeveralMatch}\\
SatisfySetterMethod\\
\hline
m \in pvCharacteristics\\
dataProviders? = \{pvCharacteristicsReader, pvExactAllocationReader\}\\
outputid = \{pvCharacteristicsReader \mapsto dataIdPvCharacteristics,\\
\qquad\qquad pvExactAllocationReader \mapsto dataIdPvCharacteristics\}\\
\hline
\end{array}$$

$$\begin{array}{l}
1 : \langle\!| m == \langle\!| name == 35, argType == 14, value == 9, optional == 0 |\!\rangle,\\
m' == \langle\!| name == 35, argType == 14, \mathbf{value} == \mathbf{8}, optional == 0 |\!\rangle,\\
dataProviders? == \{14, 17\}, outputid == \{(14, 35), (17, 35)\},\\
matching! == \{14, 17\}, \mathbf{value!} == \mathbf{8} |\!\rangle
\end{array}$$

The remaining two cases are similar. This validation approach uses the set-oriented nature of Z to check that each of the four test cases is correctly defined (no inconsistencies, and no missing/unspecified values), and that each test of an operation returns a unique solution (a singleton set) that contains the expected results. This gives us strong confidence that the specified operation is deterministic and correct, and that it has the four behaviours that we desire.

7 Reflections on Validation with ZLive

ZLive can be used in a fully automatic mode to evaluate the truth of all the conjectures in a specification, or in an interactive mode where we investigate the solutions to specific schemas one by one. In general, each ZLive evaluation has three possible outcomes: a solution is returned, or '`no more solutions`' is reported, or an evaluation error states that part of the evaluation is too large/infinite to be able to continue. In the latter two cases, we found it is often useful to use the ZLive '`why`' command to find out exactly why the evaluation failed. This prints the sorted predicates of each set comprehension, with a 'high-tide' marker just after the furthest predicate that was successfully evaluated.

For example, when the *PvAssetReaderFactoryIsValid* conjecture initially failed in the previous section, the '`why`' command showed a large set comprehension that contained:

```
    tmp3672 = 30 .. 39;
    tmp3671 = P tmp3672;      %% powerset
    . . .
    tmp3706 = { 17, 14 };
    consumes = tmp3706;
    %%---------------                (The high-tide marker)
    consumes in tmp3671 ;
```

which made it clear that although the *consumes* set was fully evaluated to $\{17, 14\}$, it failed its type test: $consumes \in \mathbb{P}(30..39)$ (because this is the first line after the high-tide marker).

A more interesting example is if we accidentally write $s.name = noDataId$ for the second-to-last predicate in the *GetMissingDependencies* in Section 5. In that case, when we evaluate the following test schema:

$$testGetMissing == [GetMissingDependencies \mid modules = EgModulePvAsset; \\ factories? = PvAssetFactory;\ \#\, missing! = 1]$$

ZLive just reports no more solutions. But the '`why`' command shows:

```
tmp2654 = {   . . .
    tmp2911 = {
        f in factories? :: 0.2097 ;
        m in modules :: 0.3276 ;
        . . .   @ unsatisfied
    };
    missing! = tmp2911;
    missing! in tmp2907 :: 1024.0 ;
    # missing! = tmp3197;
    %%---------------
    tmp3197 = 1;
    tmp3199 = <| modules==modules, outputid==outputid, modules'==modules',
            outputid'==outputid', factories?==factories?, missing!==missing! |>
  @ tmp3199
  }
%%----------
```

These two high-tide marks tell us that the outer set comprehension succeeded (giving the empty set, hence '`no more solutions`'), but in the inner set it was the cardinality of *missing*! set that caused the failure. Removing this cardinality check from *test*1 then shows us that *missing*! was empty, and a further '`why`' command shows us that potential members of *missing*! were found, but they were *not* in *f.members*, which helps to expose the error in the specification. So the '`why`' command is quite helpful for debugging specifications that fail to return the expected results.

8 Conclusions

We have specified and validated key aspects of the MODAM module manager, which automates the building of models from a set of components. The set-based validation approach described here extends previous work on the unit testing of Z specifications [9] by using a mixture of automated and interactive animation, and in using the '`why`' command to help debug set comprehensions.

In this case study, the Z specification was written after the Java implementation of the module manager, because the module manager was quite complex, and we wanted to understand its abstract behaviour more clearly. The process of specifying it in Z exposed several places where the Java implementation could have been simplified, and a couple of potential errors in the Java implementation where it omitted semantic checks that could have allowed undefined behaviour on erroneous inputs (e.g. circular dependencies between modules). Using ZLive to validate the Z specification exposed several minor errors in the Z specification, as discussed above. Difficulties with animating some schemas also uncovered one bug in ZLive that caused some disjunctive predicates to generate an error rather than evaluating correctly, and led to two minor improvements in the animation algorithms. These benefits are a good return for around 2.5 expert person days to write and debug the Z, plus another 1.5 days spent improving ZLive.

The most similar tool to ZLive is the Pro-B animator for B [6]. This is more sophisticated than ZLive, can animate multiple input languages, and has several user interfaces. Like ZLive, it offers automatic and interactive modes, can handle some infinite constructs, and can be used to generate solutions to a specification, to find counter-examples, or to search for simple proofs via finite model-checking. However, it does not seem to have any command corresponding to the ZLive '`why`' command, which allows a sophisticated user to see how far it got through animating each nested set comprehension within the specification.

References

1. 13568, I.: Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics. ISO/IEC (2002), first Edition 2002-07-01
2. Boulaire, F., Utting, M., Drogemuller, R.: Dynamic agent composition for large-scale agent-based models. Complex Adaptive Systems Modeling 3(1) (2015)
3. Cai, C., Jahangiri, P., Thomas, A.G., Zhao, H., Aliprantis, D.C., Tesfatsion, L.: Agent-based simulation of distribution systems with high penetration of photovoltaic generation. In: Power and Energy Society General Meeting. IEEE (2011)
4. Forum, C.F.G.: Change and choice. Report, CSIRO (01/12/2013 2013)
5. Klügl, F., Bazzan, A.L.C.: Agent-based modeling and simulation. AI Magazine 33(3), 29–40 (2012)
6. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. pp. 855–874. LNCS 2805, Springer-Verlag (2003)
7. North, M., Conzelmann, G., Koritarov, V., Macal, C., Thimmapuram, P., Veselka, T.: E-laboratories: agent-based modeling of electricity markets (2002)
8. North, M.J.: A theoretical formalism for analyzing agent-based models. Complex Adaptive Systems Modeling 2(1), 3–3 (2013)
9. Utting, M., Malik, P.: Unit testing of Z specifications. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ. LNCS, vol. 5238, pp. 309–322. Springer (2008)
10. Weidlich, A.: Engineering interrelated electricity markets: an agent-based computational approach. Springer [distributor], Heidelberg (2008)