Chapter 4
Agent-Based Computational Demography and Microsimulation Using JAS-mine

Matteo Richiardi and Ross E. Richardson

4.1 Introduction

In this chapter, we present the implementation of a dynamic microsimulation with a rich set of demographic processes (birth, death, household formation and dissolution) and other life course events (educational choices, labour market participation and employment outcomes), using the recently upgraded JAS-mine simulation platform (www.jas-mine.net).

The chapter is meant to provide a step-by-step guide to the development of dynamic microsimulations/agent-based models. From a practical perspective, the model presented here is highly reusable and can be easily modified in order to develop other microsimulation/agent-based models.[1] This is thanks to the JAS-mine architecture, which envisages a neat separation between data (parameters

[1]The model and the supporting documentation can be downloaded from the demo section of the JAS-mine website (www.jas-mine.net/demo/demo07).

M. Richiardi (✉)
Institute for New Economic Thinking at the Oxford Martin School, Mathematical Institute, University of Oxford, Oxford, UK

Nuffield College, Oxford, UK

Collegio Carlo Alberto, Moncalieri, Italy

Department of Economics and Statistics, University of Torino, Turin, Italy
e-mail: matteo.richiardi@maths.ox.ac.uk; matteo.richiardi@unito.it

R.E. Richardson
Institute for New Economic Thinking at the Oxford Martin School, Mathematical Institute, University of Oxford, Oxford, UK
e-mail: ross.richardson@maths.ox.ac.uk

A. Grow, J. Van Bavel (eds.), *Agent-Based Modelling in Population Studies*,
The Springer Series on Demographic Methods and Population Analysis 41,
DOI 10.1007/978-3-319-32283-4_4

and coefficients) and code, and a clear distinction between *modelling objects*, which specify the structure of a model and should be the primary concern of a researcher, and *auxiliary objects*, which perform useful tasks such as input-output communication, real-time visualization, etc.

The chapter is structured as follows. Section 4.2 motivates the need for a unique platform for agent-based models and dynamic microsimulations, integrating tools used by both modelling approaches. The section also lists other requirements that were specified for the platform. Section 4.3 briefly describes the technical solutions that were adopted to meet these requirements. Section 4.4 presents the general structure of a JAS-mine project. Section 4.5 describes the specific simulation model implemented. Section 4.6 illustrates the JAS-mine implementation and Sect. 4.7 concludes.

4.2 Convergence Between Agent-Based and Microsimulation Models

Microsimulation is a technique used in a large variety of fields to simulate the states and behaviours of different units (individuals, households, populations, etc.) as they evolve in a given environment (a market, a region, an institution). The word "dynamic" refers to the fact that the population being simulated is also changing, as opposed to "static" microsimulations (such as tax and benefit simulators) which examine the impact of a policy change on a given population (Li and O'Donoghue 2012). The modelling of demographic processes is therefore the salient characteristic of dynamic microsimulations.

Agent-based models are also computational models with individuals as the primary object of analysis. They mainly differ from microsimulations for their emphasis on the role of interaction and for explicit departures from the standard assumptions of economic models: rational expectations, perfect knowledge about the environment, infinite computational ability, absence of centralised "top down" coordination devices (Richiardi 2012).

Agent-based (AB) and microsimulation (MS) models share many features and can be described as belonging to the same class of discrete-event simulations. Indeed, from a mathematical and computational perspective the two approaches are identical. Both AB and MS models are recursive models, where the number and individual states of the agents in the system are evolved by applying a sequence of algorithms to an initial population (Gilbert and Troitszsch 2005). As computer-based simulations, they face the problem of reproducing real-life phenomena, many of which are temporally continuous processes, using discrete microprocessors. The abstract representation of a continuous phenomenon in a simulation model requires that all events be presented in discrete terms, hence the label discrete-event simulation.

However, in their historical development AB models and microsimulations have followed different trajectories (Richiardi 2013): AB models have focused more on

theory, while MS models have evolved as more data oriented, with the processes generally specified as probabilistic regression models. As a generalisation, AB models are structural models with a primary concern on *understanding*, while microsimulations are reduced-form models (as such, they often focus on one side of a market only), with a primary concern on *forecasting*.[2] However, a trend towards a convergence of the two approaches is currently underway, with AB models becoming increasingly empirically oriented, and MS models including more feedback effects (see again Richiardi (2013)). An example of this fruitful integrated approach is the recent field of agent-based computational demography (Billari and Prskawetz 2003).

The differences in scope and perspective between the two approaches have, however, impinged on the structure of the computer models used within each community. AB models lead naturally to an explicit object-oriented representation, while MS models are generally built around a database which is evolved forward in time. This has led to the development of simulation toolkits which are specific to each field, as for instance NetLogo (Wilenski 1999), RePast (North et al. 2013) and Mason (Luke et al. 2005) for AB modelling, and Modgen (Statistics Canada 2009), LIAM2 (De Menten et al. 2014) and JAMSIM (Mannion et al. 2012) for MS modelling – to name just a few.

JAS-mine was created to make the development of "hybrid" AB-MS models easier, and to allow researchers to use the same tools for both approaches, to exploit economies of scale in learning and coding. Its combination of features distinguish it from all the above platforms.

4.3 The JAS-mine Architecture

JAS-mine is an object-oriented Java-based platform for discrete-event simulations. The philosophy of JAS-mine is to foster *clarity*, *transparency* and *flexibility*. The rationale behind this is the belief that a major bottleneck in agent-based and dynamic microsimulation modelling comes from humans, rather than machines: minimizing modelling time then becomes even more important than minimizing computing time.[3] To this aim, a strict adherence to the open source paradigm was enforced

[2]Structural models often include unobservable parameters that help describe individual behaviour at a deep level (say, in terms of utility maximisation); reduced-form models aim more simply at identifying statistical relationships between observable characteristics.

[3]The performances of JAS-mine with respect to speed of execution, though, are noteworthy. An exercise aimed at testing the performance of the simulation platform with respect to scaling involved the implementation in JAS-mine of a complex mixed AB-microsimulation model of the two-way relationship between health and economic inequality, calibrated on both US and Canadian cities. The JAS-mine implementation can run five million agents with a time-step equivalent to 1 day for 500 years (182,500 time-steps) in 50 min on a standard laptop (using less than 4GB of RAM).

in the design of the platform, which makes it less of a black-box with respect to proprietary software and encourages cooperative development of the platform by the community of users: all functions can be inspected and, if necessary, modified or extended. Also, it was decided not to develop an ad-hoc grammar and syntax – as in NetLogo and LIAM2 – but to allow the user to choose from a wide range of classes and interfaces which extend the standard Java language. The JAS-mine libraries therefore provide open tools to "manufacture" a simulation model, making use whenever possible of solutions already available in the software development community. This ensures efficiency and a maximum amount of flexibility in model building.

In the platform architecture, a clear distinction is made between objects with a modelling content, which specify the structure of the simulation, and objects which perform useful but auxiliary tasks, from enumerating categorical variables to building graphical widgets, from creating filters for the collection of agents to computing aggregate statistics to be saved in the output database. Moreover, a separation is made between code and data, with all parameters and input tables stored either in an input database or in specific MS Excel files. For instance, the *regression* package provides tools for simulating outcomes from standard regression models (OLS, probit/logit, multinomial probit/logit): in particular, there is no need to specify the variables that enter a regression model, as they are directly read from the data files. This greatly facilitates exploration of the parameter space, testing different econometric specifications, and scenario analysis.

From a modelling viewpoint, JAS-mine extends the *Model-Observer* paradigm introduced by the Swarm experience (Minar et al. 1996; Luna and Stefansson 2000) and introduces a new layer in simulation modelling, the *Collector*. The Model deals mainly with specification issues, creating objects, relations between objects, and defining the order of events that take place in the simulation. The Observer allows the user to inspect the simulation in real time and monitor some pre-defined outcome variables as the simulation unfolds. The Collector collects the data and computes the statistics needed both by the simulation objects and for post-mortem analysis of the model outcome, after the simulation has completed. This three-layer methodological protocol allows for extensive re-use of code and facilitates model building, debugging and communication.

As for input/output (I/O) communication, building on the vast number of software solutions available, JAS-mine allows the user to separate data representation and management from the implementation of processes and behavioural algorithms. The management of input data persistence layers and simulation results is performed using standard database management tools, and the platform takes care of the automatic translation of the relational model of the database into the object-oriented simulation framework, through object-relational mapping (ORM).[4] This also allows

[4]ORM is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

to separate data creation from data analysis. As the statistical analysis of the model output is possibly intensive in computing time, performing it in real time might be an issue in large-scale applications. A common solution is to limit it to a selected subset of output variables. This, however, requires identifying the output of interest before the simulation is run. If additional computations are required to better understand how the model behaves, the model has to be run again: the bigger the model, the more impractical this solution is. On the other hand, the power of modern relational database management systems (RDBMS) makes it feasible to keep track of a much larger set of variables, for later analysis. Also, the statistical techniques envisaged, and the specific modeller's skills, might suggest the use of external software solutions, without the need to integrate them in the simulation machine. Finally, keeping data analysis conceptually distinct from data production enhances brevity, transparency and clarity of the code.

The architectural characteristics of JAS-mine are discussed in detail on the JAS-mine website (www.jas-mine.net). To summarise, the main features of the platform are:

- a discrete-event simulation engine, allowing for both discrete-time and continuous-time simulation modeling;[5]
- a *Model-Collector-Observer* structure (see Sect. 4.4);
- interactive (GUI based), batch and multi-run execution modes, the latter allowing for detailed design of experiments (DOE);
- a library implementing a number of different matching methods, to match different lists of agents;
- a library implementing a number of different alignment methods, to force the microsimulation outcomes to meet some exogenous aggregate targets (Li and O'Donoghue 2014);
- a library implementing a number of common econometric models, from continuous response linear regression models to binomial and multinomial logit and probit models;
- a statistical package based on the *cern.jet* package;
- an embedded H2 database;
- MS Excel I/O communication tools;

[5]Discrete-event simulations can be organized into two categories, depending on how time is treated. *Discrete-time* simulations break up time into regular time slices (Δt), while the simulator calculates the variation of state variables for all the elements of the simulated model between one point in time and the next. Nothing is known about the order of the events that happen within each time period: discrete events (marriage, job loss, etc.) could have happened at any moment in Δt while inherently continuous events (aging, wealth accumulation, etc.) are often thought to progress smoothly between one point in time and the next. By contrast, *continuous-time* simulations are characterized by irregular timeframes that are punctuated by the occurrence of the events. What is modelled is not whether an event occurs or not in the reference period, but rather the time elapsed before its occurrence (duration models). Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next. Inherently continuous events must therefore be discretised (Keller et al. 1993).

- automatic GUI creation for parameters by using Java annotation;
- automatic output database creation;
- automatic agents' sampling and recording;
- powerful probes for real-time statistical analysis and data collection;
- a rich graphical library for real-time plotting of simulation outcomes;
- Eclipse plugin, which enables the creation of a JAS-mine project in just a few clicks, with template classes organised in the JAS-mine standard package and folder structure;[6]
- Maven version control.

4.4 The Structure of a JAS-mine Project

In the JAS-mine architecture, agents are organized and managed by components called *managers*. As already mentioned, there are three types of managers in this architecture: *Model*, *Collector* and *Observer*. Models serve to build artificial agents and objects, and to plan the time structure of events. Collectors are managers that build data structures and routines to calculate (aggregate) statistics dynamically, and that build the objects used for data persistence. The definition of a Collector's schedule specifies the frequency of statistics updating and agent sampling, and consequent storage in the output database. Observers are managers that serve to build graphical widget objects that indicate the state of the simulation in real time, and define the frequency with which to update these objects.

JAS-mine allows multiple Models (and multiple Collectors and Observers) to run simultaneously, since they share the same scheduler.[7] This allows for the creation of complex structures where agents of different Models can interact. Each Model is implemented in a separate Java class that creates the objects and plans the schedule of events for that Model. Model classes require the implementation of the SimulationManager interface, which implies the specification of a *buildObjects* method to build objects and agents, and a *buildSchedule* method for planning the simulation events. Analogously, Collector classes must implement the CollectorManager interface, and Observer classes must implement the ObserverManager interface.

When a new JAS-mine project is created using the JAS-mine Eclipse plugin, several packages are created:

- **data**: package containing the classes that describe the structure of coefficients, parameters and agent population tables contained in the database to be loaded by

[6]Eclipse Integrated Development Environment is a software application that provides support to aid software development. A description of how to start using Eclipse and the JAS-mine plugin can be found at http://www.jas-mine.net/how-to-create-and-run-a-new-jas-project-using-eclipse.

[7]Technically, the scheduler is a "singleton". In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to one object.

the ORM. When using Excel files to specify input data, no specific classes need to be included in this package.

- **model**: package containing the classes that specify the model structure; in particular, it contains the Model manager class(es) and the class(es) of agents that populate the simulation.
- **model.enums**: subpackage containing the definition of the enumerations used (if any).[8]
- **experiment**: package containing the classes that deal with running the simulation experiment(s); it contains, in particular, the Start class where the main method and the type of the experiment (interactive vs. batch mode, single run vs. multiple runs) are defined. The package might also contain one or more Collectors, who compute statistics and persist the output in the database, one or more Observers for online statistics collection and display, and a MultiRun class that manages repeated runs for parameter exploration.
- **algorithms**: package containing classes that implement algorithms for determining events and applying processes to the agents. These implementations, in a cooperative effort of users, are potential candidates to extend the set of standard functions included in the JAS-mine libraries.

In addition to sources, the project also contains two folders for data input-output. The input folder contains input data in excel or H2 embedded formats. The output folder contains the output of different simulation experiments. At the beginning of each run, JAS-mine creates a sub-folder that is labelled automatically[9] with a copy of the input files plus an empty output database, with the same structure of the input database as defined by the annotations added to the model classes. Coherence between the input database (if any), the output database and the classes representing the agents in the simulation (known as *entity* classes) is guaranteed by the ORM.

By default, JAS-mine executes the simulations in embedded mode: the databases are modified directly by the JDBC driver included in JAS-mine.[10] The standard database uses a H2 database engine. Other databases supporting embedding can be used, such as Microsoft Access, Hypersonic SQL, Apache Derby, etc.

[8]Enumerations specify a set of predefined values that a property can assume. These values might be categorical (strings, e.g. gender), quantitative (discrete numbers, e.g. age) or even objects with their set of characteristics and properties (e.g. a predefined set of banks to which a firm can be linked). The ORM detects that a value is an enumeration when the property is declared with the annotation *@Enumerated* (see Sect. 4.6.3.1). Through enumerations the ORM automatically manages reading/writing operations in both text and numerical format.

[9]The folder name can be modified dynamically through labels set by the user.

[10]A JDBC driver is a software component enabling a Java application to interact with a database.

4.5 The Dynamic Microsimulation Model

The model that we implement is inspired by the *Demo07* sample model included in LIAM2.[11] It features a population of 20,200 persons grouped in 14,700 households undergoing a number of demographic and other life course events on an annual basis between the years 2002–2061:

- **Birth**: all women aged between 15 and 50 (inclusive) in any simulation year can give birth to a child, with a probability which is year- and age-specific and is reported in the file *p_birth.xls*.
- **Education**: education (lower secondary, upper secondary or tertiary) is predetermined at birth. Individuals exit lower secondary education at age 16, upper secondary education at age 19, and tertiary education at age 24.
- **Exit from parental home**: individuals aged 24 or over who are not yet married leave their parental home to set up a new household.
- **Marriage**: all individuals aged 18 or over whose civil state is either single, divorced or widowed, are eligible for getting married in any given simulation year. The probability of marriage depends on age, gender and civil state, and is stored in the *p_marriage.xls* file. Given these probabilities, a subset of the unmarried population is sampled and those chosen are entered into the matching algorithm. Actual matching involves ordering all the females first; then starting with the top ranked female, all males are ordered and the best available male is matched. Then for the second ranked female, the remaining males are ordered and the best available male is matched, and so on until no more matches can be made (because there are either no more males or females to match). Females are ordered according to their age difference (in absolute value) with respect to the average age in the pool of females to be matched, $|age - \text{mean}(age)|$: females with an age closer to average 'choose' first, while older or younger females 'choose' later. For each female, males are ranked by looking at how their age and work status compares with the female's age and work status: regression coefficients are stored in the *reg_marriage.xls* file. Matched couples form new households.
- **Divorce**: divorce probability depends on age difference between the partners, elapsed marriage duration, number of children and work status of both partners: regression coefficients are stored in the *reg_divorce.xls* file.

[11] The model differs from the LIAM2 version in that it collapses the work states of unemployment and inactivity into a single non-employment state. This is done by removing the unemployment module from the corresponding LIAM2 simulation, with everything else staying the same. The change is motivated by the fact that the distinction between unemployment and inactivity was implemented in a very unnatural way in LIAM2, and did not affect any subsequent choice on the part of the agents.

- **Employment**: all individuals who are of working age (males: between 15 and 65; females: between 15 and 61) and whose previous work state was neither student nor retired are considered to be available to work. Conditional on this, individuals are employed with a probability which depends on age, lagged work state (either employed, unemployed or inactive), gender and marital status: regression coefficients are stored in the *reg_inwork*.xls file. The model does not distinguish between unemployment and non-employment.
- **Death**: death is also a probabilistic event, with year- and age-specific death probabilities contained in the files *p_death_m.xls* and *p_death_f.xls*, for males and females respectively.

The divorce and employment processes are subject to alignment. This is a technique widely used in (dynamic) microsimulation modelling to ensure that the simulated totals conform to some exogenously specified targets, or aggregate projections (Baekgaard 2002; Klevmarken 2002; Li and O'Donoghue 2014). Alignment targets (aggregate frequencies) for divorce and employment are stored in the *p_divorce.xls* and *p_inwork.xls* files respectively.

One important thing to note is that the processes to be aligned are executed at an individual level, while alignment always takes place at the population level. That is, individual outcomes or probabilities are determined for each individual based on the chosen econometric specification and the estimated coefficients. This in general leads to a mismatch between the simulated (provisional) totals and the aggregate targets, which can of course be assessed only at the population level. The alignment algorithm then directly modifies the individual outcomes or probabilities of occurrence.

The specific algorithm used in the LIAM2 implementation is called "Sorting by the difference between the predicted probability and a random number" (SBD), see Li and O'Donoghue (2014), and – though quite common in the microsimulation literature – can be criticised on many theoretical grounds, see Stephensen (2014). The JAS-mine *alignment* library implements it for completeness, though its use is deprecated. Here we use it to remain as close as possible to the original LIAM2 version (the reader does not need to understand precisely how it works).

Finally, note that, though agents' interactions are limited to matching in the marriage market, the model contains all the basic ingredients of a standard agent-based model. Further "agentisation" could entail introducing more interaction in the labour market, or introducing competition for instance in residential locations.[12]

[12]The interested reader will find a JAS-mine implementation of the Schelling Segregation Model, with added microsimulation features for illustrative purposes (a dynamic population, with birth, ageing and death processes) in the demo section of the JAS-mine website (www.jas-mine.net/demo/extended-schelling).

4.6 The JAS-mine Implementation

The JAS-mine class structure of the *Demo07* model is organised as in Table 4.1.

The core of the simulation lies in the *model* package, which contains the classes PersonsModel, Person and Household. The *experiment* package contains the StartPersons class that specifies to run the simulation in interactive mode, the PersonsCollector class that collects all the statistics needed to monitor the simulation and updates the output database, and the PersonsObserver class that creates and manages the graphical object for runtime monitoring. Parameters and coefficients are loaded into the Parameters class in the *data* package. All filters used to filter collections are grouped in the *data.filters* subpackage. The categories used for gender, educational levels, civil state and work state are stored as Enums in the *model.enums* subpackage. Finally the *algorithms* package contains classes that perform technical tasks (in the example, MapAgeSearch searches age- and gender-specific values in a map of coefficients, with age and gender as keys). Classes in the *algorithms* package are meant to be of general use beyond the specific model being developed, and are candidates for inclusion in the core libraries in future releases of the platform.

The project is organised in the standard JAS-mine folder structure, as described in Table 4.2.

The Java classes are contained in the *src* folder. The *input* folder contains the MS Excel parameter files and the initial population, stored as an h2 database (input.h2). The *output* folder is initially empty. For each new simulation experiment, a new subfolder is created with the appropriate time stamp, so as to uniquely identify the experiment (e.g. 20141218151116, for experiments initiated on the 18th of

Table 4.1 Class structure

Package	Class
Algorithms	MapAgeSearch
Data	Parameters
data.filters	ActiveMultiFilter
	FemaleFilter
	FemaleToCoupleFilter
	FemaleToDivorce
	MaleFilter
	MaleToCoupleFilter
Experiment	PersonsCollector
	PersonsObserver
	StartPersons
Model	Household
	Person
	PersonsModel
model.enums	CivilState
	Education
	Gender
	WorkState

Table 4.2 File structure

Folder	Files	Notes
Input	p_birth.xls	Birth probabilities, by age and (simulated) year
	p_death_f.xls	Death probabilities, by age and (simulated) year, for females
	p_death_m.xls	Death probabilities, by age and (simulated) year, for males
	p_marriage.xls	Marriage probability, by age group, gender and civil state
	p_divorce.xls	Alignment target for the divorce probability, by age group and (simulated) year
	p_inwork.xls	Alignment target for the employment probability, by age group, gender and (simulated) year
	reg_marriage.xls	Marriage score coefficients: determine how well a specific male fits a given female
	reg_divorce.xls	Divorce coefficients: determine the (unaligned) probability of divorcing
	reg_inwork.xls	Employment coefficients: determine the (unaligned) probability of being employed
	input.h2	Initial population
Output	(Empty)	
Src	(All java classes)	See Table 4.1
Target	(Compiled classes)	
Libs	(External libraries and JARs, empty)	
(Root)	pom.xml	Maven version control

December 2014, at 16 s after 3.11 pm). The subfolder contains a copy of all the input files (in the *input* directory) and an output database (out.h2, in the *database* directory).

The *target* and *libs* folders contain technical content of no immediate interest to the modeller. The root folder also contains a *pom* (project object model) file, which contains information on the JAS-mine version used for the project. Apache Maven, an open source software project management and comprehension tool uses this information to manage all the project's build, reporting and documentation. In particular, by specifying in the *pom* file the desired release for each library used (including the JAS-mine libraries), Maven automatically downloads the relevant libraries from the appropriate repositories.[13]

4.6.1 Parameters

As JAS-mine supports a clear distinction between modelling classes and data structures, parameters are loaded and stored in a specific class, called Parameters.

[13]This implies that each JAS-mine project has its own copy of all the libraries used, ensuring that the project is self-contained and that it keeps working exactly as intended even when new versions of the libraries are released (and even if backward compatibility is not respected).

The class makes use of the ExcelAssistant.*loadCoefficientMap*() method to read the parameters from MS Excel files: this requires to specify a .xls file, a worksheet name, the number of key columns and the number of value columns.[14] Parameters are then stored in MultiKeyCoefficientMap objects, which are basically standard Java maps with multiple keys (Box 4.1).

Box 4.1 The Parameters.*loadParameters*() Method

```
public static void loadParameters() {

// probabilities
pBirth = ExcelAssistant.loadCoefficientMap("input/
p_birth.xls", "Sheet1", 1, 59);

pDeathM = ExcelAssistant.loadCoefficientMap("input/
p_death_m.xls", "Sheet1", 1, 59);

pDeathF = ExcelAssistant.loadCoefficientMap("input/
p_death_f.xls", "Sheet1", 1, 59);

pMarriage = ExcelAssistant.loadCoefficientMap
("input/p_marriage.xls", "Sheet1", 3, 4);

pDivorce = ExcelAssistant.loadCoefficientMap
("input/p_divorce.xls", "Sheet1", 2, 59);

pInWork = ExcelAssistant.loadCoefficientMap("input/
p_inwork.xls", "Sheet1", 3, 59);

// regression coefficients
coeffMarriageFit = ExcelAssistant.loadCoefficientMap(
                   "input/reg_marriage.xls","Sheet1",1, 1);

coeffDivorce = ExcelAssistant.loadCoefficientMap(
               "input/reg_divorce.xls", "Sheet1", 1, 1);

coeffInWork = ExcelAssistant.loadCoefficientMap(
              "input/reg_inwork.xls", "Sheet1", 3, 1);

// definition of regression models
regMarriageFit = new LinearRegression(coeffMarriageFit);

regDivorce = new LogitRegression(coeffDivorce);

regInWork = new LogitRegression(coeffInWork);

}
```

[14]It is also possible to load the parameters from a table in the input database. See the online documentation for further details.

There are two types of parameters in *Demo07*: probabilities and regression coefficients.

Birth (*pBirth*) and death (*pDeathM* and *pDeathF*) probabilities have one key (age), while the value columns refer to different simulation years: birth and death probabilities are therefore age- and year-specific.

Divorce probabilities (*pDivorce*) have two keys (the lower and upper bounds defining age groups), while value columns refer again to different simulation years: divorce probabilities are therefore age group- and year-specific.

Marriage (*pMarriage*) and employment (*pInWork*) probabilities have three keys (the lower and upper bounds defining age groups and gender). Value columns in *pMarriage* refer to different civil states: marriage probabilities are therefore age group-, gender- and civil state-specific. Value columns in *pInWork* refer to different simulation years: employment probabilities are therefore age group-, gender- and year-specific.

Table 4.3 shows how the *p_birth.xls* file looks like.

Regression coefficients can have one key (*coeffMarriageFit* and *coeffDivorce*) which is the regressor variable name, and a corresponding value with the estimated coefficient. They might have additional keys, as in *coeffInWork*, if the coefficients are differentiated by some other variables (gender and employment state, in this example). Table 4.4 shows what the corresponding *reg_inwork.xls* file looks like.

Table 4.3 Extract from the *p_birth.xls* file

Simulation year			
Age	2002	...	2060
15	0.00068	...	0.00075
16	0.00186	...	0.00181
...			
50	0.00010	...	0.00021

Table 4.4 Extract from the *reg_inwork.xls* file

Regressors	Gender	workState	Coefficients
Age	Male	Employed	−0.19660
isMarried	Male	Employed	0.18928
workIntercept	Male	Employed	3.55461
...			
Age	Male	NotEmployed	0.97809
workIntercept	Male	NotEmployed	−12.39108
...			
Age	Female	Employed	−0.27405
isMarried	Female	Employed	−0.09068
workIntercept	Female	Employed	3.64871
...			
Age	Female	NotEmployed	0.82176
isMarried	Female	NotEmployed	−0.55910
workIntercept	Female	NotEmployed	−10.48043
...			

Note that the name of the regressor variable must appear in the first column, as the regression classes expect it to be the first key in the MultiKeyCoefficientMap instance. The name of the headings for the additional key columns must match the name of a field in the relevant class, in this case, the Person class.

The appropriate regression models are then defined based on the regression coefficients.

4.6.2 The PersonsModel Class

4.6.2.1 Objects

The Model extends the AbstractSimulationManager class. This requires implementing the *buildObjects*() and the *buildSchedule*() methods. The *buildObjects*() method contains the instructions to create all the agents and the objects that represent the virtual environment for model execution (see Box 4.2).[15] In *Demo07*, this involves loading the parameters for the simulation and the initial population, made of persons and households. Three other methods complete the simulation setup: *initializeNonDatabaseAttributes*() initializes attributes that do not appear in the input database, such as the education level; *addPersonsToHouseholds*() registers household members, and *cleanInitialPopulation*() checks the internal consistency of the initial population and removes errors, making sure that all marriage partnerships are bilateral, that all partners belong to the same household, and that no empty households exist.[16]

Box 4.2 The PersonsModel.*buildObjects*() Method

```
@Override
public void buildObjects() {

Parameters.loadParameters();

persons = (List<Person>) DatabaseUtils.loadTable
(Person.class);

households = (List<Household>)
DatabaseUtils.loadTable(Household.class);

initializeNonDatabaseAttributes();
```

(continued)

[15]The *@Override* annotation is used by the Java interpreter to ensure that the programmer is aware that the method declared is overriding the same method declared in the superclass.

[16]This method is absent in the LIAM2 implementation, which does not get rid of all the errors in the initial database.

```
addPersonsToHouseholds();

cleanInitialPopulation();

}
```

As we have seen, the general rule is that parameters should not be hard-coded in the simulation. The only exception is with *control parameters* that can be changed from the GUI before the simulation starts or while the simulation is running in order to experiment with the model behaviour in interactive mode. Control parameters are properties of a simulation, they are annotated with *GUIparameter*, are automatically loaded into the JAS-mine GUI, and are automatically saved in a separate table of the output database. In *Demo07* there are just three such parameters, as described in Box 4.3.

Box 4.3 PersonsModel: Control Parameters

```
GUIparameter(description="Simulation begins at
year [valid range 2002-2060]")
private Integer startYear = 2002;

GUIparameter(description="Simulation ends at year
[valid range 2003-2061]")
private Integer endYear = 2061;

GUIparameter(description="Retirement age for women")
private Integer wemra = 61;
```

4.6.2.2 Schedule

The *buildSchedule*() method contains the plan of events for the simulation. Events are planned based on a discrete event simulation paradigm. This means that events can be scheduled dynamically at specific points in time. The frequency of repetition of an event can be specified in the case of periodic events. An event can be created and managed by the simulation engine (a system event e.g. terminating the simulation), it can be sent to all the components of a collection or list of agents or it can be sent to a specific object/instance. Events can be grouped together if they share the same schedule.

In *Demo07*, all events are scheduled right from the beginning of the simulation (there is no dynamic scheduling), and occur on a yearly basis. They are grouped in an EventGroup called *modelSchedule*, which is scheduled at every simulation

period starting at startYear with the *scheduleRepeat*(Event *event*, double *atTime*, int *withOrdering*, double *timeBetweenEvents*) method:

```
getEngine().getEventList().scheduleRepeat(modelSchedule,
startYear, 0, 1.);
```

The events of *Demo07* are typically directed to a collection of objects – persons or households – and are inserted into an EventGroup with the instruction

```
modelSchedule.addCollectionEvent(Object object, [some action the
object must perform]);
```

The actions to be performed can be specified in two ways. The simplest is to use Java reflection and simply specify the object's method name to be invoked.[17] For instance, asking all persons to perform the *ageing*() method would require the instruction:

```
modelSchedule.addCollectionEvent(persons, Person.class,
"ageing");
```

Java reflection, however, generally has a reputation for being quite slow. A better approach is to use the EventListener interface. When an object implements this interface, it must define an *onEvent*() method that will receive specific enumerations to be interpreted. We will describe how the Person class implements the *onEvent*() method in Sect. 4.6.3.3. For now, we simply note that by using the EventListener interface, the scheduling of the *ageing*() method becomes:

```
modelSchedule.addCollectionEvent(persons, Person.Processes.
Ageing);
```

By default, the broadcasting of an event to a collection of objects is performed in *safe mode* (read only), and does not allow the concurrent modification of the collection itself. This is not a problem with the *ageing*() process, as ageing per se does not entail any modification in the list of persons, that is, it does not add or remove anyone. This is not true with other processes, like *birth*() or *death*(). In order to allow the collection to be changed while iterated by the simulation engine, this feature has to be switched off, as in

```
modelSchedule.addCollectionEvent(persons, Person.Processes.Death,
false);
```

The last argument specifies that the collection is subject to changes while being iterated, and the JAS-mine engine treats it accordingly.

The order of the events in the simulation follows the original LIAM2 implementation and is specified in Box 4.4: there is first a set of demographic events (ageing, death, birth, marriage, exit from parental home, divorce, household composition) and then a set of events that define the work status (whether in education, retired, other non-employed, or employed).

[17] This requires the method – *ageing*() in this case – to be declared public.

Box 4.4 The PersonsModel.*buildSchedule*() Method

```
@Override
public void buildSchedule() {

EventGroup modelSchedule = new EventGroup();
// 1: Ageing
modelSchedule.addCollectionEvent(persons, Person.Processes.
Ageing);

// 2: Death
modelSchedule.addCollectionEvent(persons, Person.Processes.
Death, false);

// 3: Birth
modelSchedule.addCollectionEvent(persons, Person.Processes.
Birth, false);

// 4: Marriage
modelSchedule.addCollectionEvent(persons, Person.Processes.
ToCouple);

modelSchedule.addEvent(this, Processes.MarriageMatching);

// 5: Exit from parental home
modelSchedule.addCollectionEvent(persons, Person.Processes.
GetALife);

// 6: Divorce
modelSchedule.addEvent(this, Processes.DivorceAlignment);

modelSchedule.addCollectionEvent(persons, Person.
Processes.Divorce);

// 7: Household composition
// (for reporting only: household composition is
// updated whenever
// needed throughout the simulation)
modelSchedule.addCollectionEvent(households,
Household.Processes.HouseholdComposition);

// 8: Education
modelSchedule.addCollectionEvent(persons, Person.Processes.
InEducation);

// 9: Work
modelSchedule.addEvent(this, Processes.InWorkAlignment);

getEngine().getEventList().schedule(modelSchedule, 0, 1);

getEngine().getEventList().schedule(
```

(continued)

```
new SingleTargetEvent(this, Processes.Stop),
endYear - startYear);

}
```

Marriage is performed in two steps. First, a subset of suitable males and females are selected for matching by invoking the method Person.*toCouple*()[18]; then, matching takes place. As we have seen in Sect. 4.5, matching uses a "centralised" algorithm and is therefore performed by the Model itself. Consequently, this event is a single target event, rather than a collection event, and is inserted into our EventGroup *modelSchedule* with the instruction

```
modelSchedule.addEvent(this, Processes.MarriageMatching);
```

Similarly, the divorce and work events are subject to alignment and are managed directly by the Model, with the methods *divorceAlignment*() and *inWorkAlignment*(), though divorce also requires some actions taken by the individuals themselves – in the *divorce*() method in the Person class – after they have been selected to divorce. *householdComposition*() is the only method which is directed to the collection of households. It simply updates the number of adults and children in each household for reporting purposes. A final single target event is scheduled for the last year of the simulation with the method *scheduleOnce*(Event *event*, double *atTime*, int *withOrdering*): its target is the Model itself and brings the simulation to a halt:

```
getEngine().getEventList().scheduleOnce
(new SingleTargetEvent(this, Processes.Stop), endYear,
Order. AFTER_ALL.getOrdering());
```

4.6.2.3 The EventListener Interface

Since the Model performs actions during the simulation, as with the Person and Household classes, it implements the EventListener interface. This requires first to enumerate all the actions that the Model is supposed to perform (this is done by defining the specific Enum Processes), and then to specify the method *onEvent*() – see Box 4.5.

[18] See Sect. 4.6.3.3.

Box 4.5 Implementation of the EventListener Interface in PersonsModel

```
public enum Processes {
        MarriageMatching,
        DivorceAlignment,
        InWorkAlignment,
        Stop;
}

@Override
public void onEvent(Enum<?> type) {
        switch ((Processes) type) {
        case DivorceAlignment:
                divorceAlignment();
                break;
        case InWorkAlignment:
                inWorkAlignment();
                break;
        case MarriageMatching:
                marriageMatching();
                break;
        case Stop:
                log.info("Model completed");
                getEngine().pause();
                break;
        }
}
```

We now dig into the matching and alignment methods performed by the Model.

4.6.2.4 The Matching Algorithm

Prior to matching, a sample of the population to marry at this time is determined randomly using the Person.*toCouple*() method. Subsequently, matching involves first ordering all the females; then, for each female starting from the top of the ranking, all males are ordered and the most suitable male is matched. This continues until there are either no more females or males to match. Females are ordered according to their age difference (in absolute value) with respect to the average age in the pool of females to be matched, $|age - \text{mean}(age)|$; the female whose age is closest to the average is ranked first. To compute this ranking, the average age of the subset of females selected for matching is required. There are a number of ways to perform this computation, which is preliminary to the application of the

matching algorithm. The one that is implemented in *Demo07* makes use of Java closures (Box 4.6).[19]

Box 4.6 Computing the Average Age for the Eligible Females in PersonsModel.*marriageMatching*()

```
final AverageClosure averageAge =
new AverageClosure() {

      @Override
      public void execute(Object input) {
            add(((Person) input).getAge());
      }
};

Aggregate.applyToFilter(getPersons(),
new FemaleToCoupleFilter(), averageAge);
```

The JAS-mine *collection* package defines an AverageClosure as a closure that receives values from objects as an input and returns the mean of these values as an output. Here, it is used to compute the average age of a given set of persons. The set is defined by applying the FemaleToCouple filter to the list of all persons, with the instruction

```
Aggregate.applyToFilter(getPersons(), new FemaleToCoupleFilter(),
averageAge);
```

The *averageAge* closure now contains the average age of all filtered females. In turn, the FemaleToCouple filter simply selects the female persons who have the *toCouple* flag switched on (Box 4.7).

Box 4.7 The FemaleToCouple Filter

```
public class FemaleToCoupleFilter
implements Predicate {

@Override
public boolean evaluate(Object object) {
      Person agent = (Person) object;
```

(continued)

[19]Technically, a *closure* is a function that refers to free variables in their lexical context. A free variable is an identifier (the identity of the person which is included in the evaluation set, in our example) that has a definition outside the closure: it is not defined by the closure, but it is used by the closure. In other words, these free variables inside the closure have the same meaning they would have had outside the closure.

```
        return (agent.getGender().equals(Gender.Female)
&& agent.getToCouple());
      }
}
```

Having the filters specified as separate classes, grouped in the separate package *data.filters*, might look cumbersome at first (and there are other ways to do this, see the online documentation) but allows to keep the core code clean while using the standard Apache Predicate approach to filtering – remember that the JAS-mine approach supports the use of existing software solutions whenever possible, and envisages to keep the specificities of the JAS-mine libraries to a minimum in order to minimise the "black box" feeling of many simulation platforms.

Matching is then performed, following the LIAM2 implementation, by making use of a simple one-way matching procedure (the agents in one collection – females in our example – choose, while the agents in the other collection – males – remain passive) implemented in the SimpleMatching class:

```
matching(collection1, filter1, comparator1, collection2,
filter2, matchingScoreClosure, matchingClosure);
```

and it is invoked as

```
SimpleMatching.getInstance().matching(...);
```

The matching method requires seven arguments:

1. **collection1**: the first collection (e.g. all individuals in the population);
2. **filter1**: a filter to be applied to the first collection (e.g. all females with the *toCouple* flag on);
3. **comparator1**: a comparator to sort the filtered collection, which determines the order that the agents in the filtered collection will be matched.
4. **collection2**: the second collection, which can be the same as *collection1* (e.g. all individuals in the population) or a different one; the two collections do not need to have the same size;
5. **filter2**: a filter to be applied to the second collection (e.g. all males with the *toCouple* flag on);
6. **matchingScoreClosure**: a piece of code that assigns, for every element of the filtered *collection1*, a double value to each element of the filtered *collection2*, as a measure of the quality of the match between every pair;
7. **matchingClosure**: a piece of code that determines what to do upon matching.

As in the computation of the average age, the use of closures – which are relatively new to the Java language – allows a great simplification of the code. While it is not required that the user knows about closures, it is interesting to understand why they are so useful. In the example, suppose that the females in the population are sorted according to some criterion, for example beauty: the prettiest woman is the first to choose a partner, the second prettiest is the second to choose etc. The

matchingScoreClosure sorts all possible mates according to some other criterion, for example wealth. Hence, the prettiest woman gets the richest man, the second prettiest gets the second richest, etc. In such a case, a comparator would suffice to order the males in the population, as the ranking is the same irrespective of the female who is evaluating them. But suppose now that the attractiveness of a man depends on the age differential between himself and the potential partner: in such a case, the ranking is specific to each woman in the population. A simple comparator would still do the job, but the comparator should be able to access the identity of the woman who is making the evaluation as an argument, which requires a lot of not-so-straightforward coding. Closures allow to bypass this technical requirement because they can pass a functionality as an argument to another method; in other words, they treat functionality as method argument, or code as data.

Closures in the *matching*() method are easier to understand when illustrated by an example: the seven arguments are listed in Box 4.8.

Box 4.8 The Matching Algorithm in PersonsModel.*marriageMatching*()

```
SimpleMatching.getInstance().matching(

// collection1: the whole population
persons,

// filter1:
new FemaleToCoupleFilter(),

// comparator1: a comparator that assigns priority to the
// individual that has a lower difficulty in matching
// (this is determined by an individual's age in relation
// to the average)
new Comparator<Person>() {

@Override
public int compare(Person female1, Person female2) {
       return (int) Math.signum(
         Math.abs(female1.getAge() -
         averageAge.getAverage()) -
         Math.abs(female2.getAge() -
         averageAge.getAverage()));
       }
},

// collection2: same as collection1
persons,

// filter2:
new MaleToCoupleFilter(),

// MatchingScoreClosure: a closure that, given a specific
```

(continued)

```
// female,
// computes for every male in the population a matching score
new MatchingScoreClosure<Person>() {

        @Override
        public Double getValue(Person female, Person
        male) {
               return female.getMarriageScore(male);

               }
},
// matchingClosure: a closure that creates a link between a
// specific female and a specific male, and sets up a new
// household.
new MatchingClosure<Person>() {

        @Override
        public void match(Person female, Person male) {

              female.marry(male);
              male.marry(female);

              }
        }
);
```

4.6.2.5 Alignment

Alignment involves comparing the *provisional* outcomes of the simulation with some external aggregate targets, and then modifying the simulation outcomes in order to match the external totals. We show how this is implemented in *Demo07* by looking at the *divorceAlignment*() method; the *inWork*() alignment method works similarly. When it comes to divorce, as in *marriageMatching*(), the focus is on females: males are passive recipients of their partners' choices. Different targets are specified for different age groups and simulated years; as we have seen in Sect. 4.6.1, these are read from the file *p_divorce.xls* and stored in the MultiKeyCoefficientMap *pDivorce* in the Parameters class. The *divorceAlignment*() method works cell by cell, that is, it aligns each age group of the population to its year-specific target: this means that the alignment algorithm is applied once for every age group (as defined in the *p_divorce.xls* parameter file). The structure of the method is therefore as follows:

- For each age group: do alignment:
  - Read target from *pDivorce*.
  - Select the relevant subgroup of married females.

- Compute, for each of the selected females, a probability to divorce that depends on the age group to which they belong.
- Select the couples that divorce by applying the SBD algorithm: each female is ranked according to the signed difference between their divorce probability and a random number uniformly distributed between 0 and 1; then, the number of couples equal to the target are selected to divorce by starting with the top ranked female and going down the ranks until the target number is reached.[20]

The MultiKeyCoefficientMap *pDivorce*, which contains the targets, has a three dimensional key: the lower and upper bounds for the age group, and the year of the simulation. The age group-specific and year-specific targets are read with the instruction reported in Box 4.9.

Box 4.9 PersonsModel.*divorceAlignment*(): Reading the Targets

```
MultiKeyCoefficientMap pDivorceMap =
Parameters.getpDivorce();

for (MapIterator iterator =
pDivorceMap.mapIterator(); iterator.hasNext();) {

        iterator.next();
        MultiKey mk = (MultiKey) iterator.getKey();
        int ageFrom = (Integer) mk.getKey(0);
        int ageTo = (Integer) mk.getKey(1);
        double divorceTarget = ((Number)
        pDivorceMap.getValue(
               ageFrom,
               ageTo,
               getStartYear() + SimulationEngine.
getInstance().getTime())).doubleValue();
                [...]
}
```

The alignment methods require four arguments:

1. **collection**: a collection of individuals whose outcome or probability of an event has to be aligned (e.g. all the population);
2. **filter**: a filter to be applied to the collection (e.g. all females selected to divorce);
3. **alignmentProbabilityClosure** or **alignmentOutcomeClosure**: a piece of code that i) computes for each element of the filtered collection a probability for the

[20]The ranking involves a stochastic component (the random number that is subtracted from the divorce probability score) in order to give individuals with a low predicted probability some chance to experience the event. As we have already noted, the SBD algorithm is quite distortive and its use is deprecated in JAS-mine; it is employed here only for consistency with the LIAM2 implementation.

event (in the case that the alignment method is aligning probabilities, as in the SBD algorithm) or an outcome (in the case that the alignment method is aligning outcomes), and ii) applies to each element of the filtered collection the specific instructions coming from the alignment method used;

4. **targetShare** or **targetNumber**: the share or number of elements in the filtered collection that are expected to experience the transition; the SBD algorithm uses *targetShare*.

Box 4.10 shows how the alignment method is implemented in *Demo07*.

Box 4.10 PersonsModel.*divorceAlignment*(): Applying the SBD Alignment Algorithm

```
new SBDAlignment<Person>().align(

      // collection:
      persons,

      // filter:
      new FemaleToDivorce(ageFrom, ageTo),

      // alignmentProbabilityClosure:
      new AlignmentProbabilityClosure<Person>() {

      // i) compute the probability of divorce
      @Override
      public double getProbability(Person agent) {
            return agent.computeDivorceProb();
      }

      // ii) determine what to do with the aligned
      probabilities
      @Override
      public void align(Person agent,
      double alignedProbability)
      {
            boolean divorce = RegressionUtils.event(
                        alignedProbability,
                        SimulationEngine.getRnd()
                        );

            agent.setToDivorce(divorce);

          }
      },

      // targetShare:
      divorceTarget

);
```

4.6.3 The Person Class

4.6.3.1 Entities

The Person class is an Entity class, as specified by the @*Entity* annotation:

```
@Entity
public class Person implements Comparable<Person>,
EventListener, IDoubleSource {
                    [...]
}
```

This implies that the class is linked to a table in the database with the same name, and that all properties which are not annotated as @*Transient* are persisted in the database, when the simulation output is saved.

Entity classes must specify a PanelEntityKey (annotated as @*Id*), which is a three-dimensional object which identifies the agent id, the simulation time and the simulation run. These three keys uniquely identify each record in the database:

```
@Id
private PanelEntityKey id;
```

The ORM expects that the field names in the database are the same as the property names in the Java class, except when a different name is specified as in

```
@Column(name="dur_in_couple")
private Integer durationInCouple;
```

Enumerations can be interpreted by the ORM both as a string and as ordinal values (0 for the first *enum*, 1 for the second, etc.), depending on how they are annotated:

```
@Enumerated(EnumType.STRING)
private WorkState workState;
```

4.6.3.2 The IDoubleSource Interface

The Person class implements the IDoubleSource interface. This interface provides a simple way of asking a class to return a specific value.[21] Similarly to the EventListener interface, it requires to declare an Enum which lists all the variables that can be queried, and the *getDoubleValue*() method for returning their value (Box 4.11). It is used by the Regression classes as a way of decoupling the regression model specification from the code: as long as a variable is enumerated in the specific

[21] As such, it is also used by JAS-mine distribution plots, see Sect. 4.6.6.

Enum called Regressors, it can be used (or removed) as a covariate in a regression model without the need to modify the code.[22]

Box 4.11 Implementation of the IDoubleSource Interface in Person

```
public enum Regressors {

// For marriage regression, check with potential
// partner's properties
       potentialPartnerAge,
       potentialPartnerAgeSq,
       potentialAgeDiff,
       inWorkAndPotentialPartnerInWork,
       notInWorkAndPotentialPartnerInWork,
       ...
// For in work regression
       age,
       ageSq,
       ageCub,
       isMarried,
       workIntercept;

}

public double getDoubleValue(Enum<?> variableID) {

switch ((Regressors) variableID) {

//For marriage regression
case potentialPartnerAge:
       return getPotentialPartnerAge();
case potentialPartnerAgeSq:
       return getPotentialPartnerAge() *
getPotentialPartnerAge();

      ...

//For work regression
case age:
       return (double) age;
case ageSq:
       return (double) age * age;
case ageCub:
```

(continued)

[22]Regression classes also have a method to read directly the values of the variables from the agent class, without the need of implementing the *IDoubleSource* interface. However, this requires that all the variables used by a regression model are defined as (possibly transient) properties in the class. This is particularly tedious when the covariates refer to another agent (such as a potential partner, or the spouse), as is common in our case.

```
        return (double) age * age * age;
case isMarried:
        return civilState.equals(CivilState.Married) ?
        1.0 : 0.0;
case workIntercept:
        return 1.0;    //Constant intercept, multiply
// regression coefficient by 1
default:
        throw new IllegalArgumentException(
                "Unsupported regressor " +
                variableID.name() + " in
                Person#getDoubleValue");
        }
}
```

4.6.3.3 Methods

The Person class implements the EventListener interface and is therefore able to be activated by the scheduler with the *onEvent*() method. The calls that a Person is able to respond to – enumerated in a specific Enum called Processes (Box 4.12) – are:

- **Ageing**: age and marriage duration are increased; work status is set to retired if retirement age is reached.
- **Death**: an age-, gender- and year-specific death probability is read from the MultiKeyCoefficientMaps *pDeathM* and *pDeathF* stored in the Parameters class; this probability is then compared with a uniformly distributed random number between 0 and 1 to determine the occurrence of the event:

```
RegressionUtils.event(deathProbability);
```

  If death occurs, the partner's status is updated to widow and the person is removed from all the lists (that is, from his/her household and from the model).
- **Birth** (applied to all females aged between 15 and 50 inclusive): an age- and year-specific probability of having a baby is read from the MultiKeyCoefficientMap *pBirth* stored in the Parameters class; then the occurrence of the event is determined in a similar fashion to the *death*() process. No multiple births such as twins can occur. Newborns are given a potential educational level that will be reached with certainty. Following the LIAM2 implementation, the person is assumed to be a student until completion of their studies (at age 16 for lower secondary education, 19 for upper secondary education, and 24 for tertiary education).

- **ToCouple** (applied to all unmarried individuals aged between 18 and 90 inclusive): this method reads an age-, gender- and civil state-specific probability of forming a partnership from the MultiKeyCoefficientMap *pMarriage* and determines whether the Boolean flag *toCouple* is set to true, to be used by the *marriageMatching*() algorithm in the PersonsModel class.
- **GetALife** (leave parental home): a new household is created if the individual is aged 24 or over, unmarried and still living in the parental household.
- **Divorce**: after divorce is decided by the Model's alignment method, partner links are broken, civil states are updated, females retain their household and males move to a newly created household.
- **InEducation**: this method examines the person's age and education level to determine whether an individual is still in education, or must exit education and enter the labour market as unemployed.

Box 4.12 The Person.Processes Enum, Defining the Processes a Person Undertakes When Activated by the Scheduler

```
public enum Processes {
        Ageing,
        Death,
        Birth,
        ToCouple,
        Divorce,
        GetALife,
        InEducation;
}
```

Other significant methods of the Person class include:

- ***getMarriageScore***(): computes the score of each male in the marriage pool, for a given female, based on a linear regression model specified by the MultiKeyCoefficientMap *regMarriageFit*; it is used by the *marriageMatching*() method in the PersonsModel class.
- ***marry***(): creates a link between the two partners and sets up a new household where they move to; it is used by the *marriageMatching*() method in the PersonsModel class.
- ***computeDivorceProb***(): computes the divorce probability, based on a logit regression model specified by the MultiKeyCoefficientMap *regDivorce*; it is used by the *divorceAlignment*() method in the PersonsModel class.
- ***computeWorkProb***(): computes the employment probability, based on a logit regression model specified by the MultiKeyCoefficientMap *regInWork*; it is used by the *inWorkAlignment*() method in the PersonsModel class.

Given that the regression coefficients have already been loaded from Excel files into the Parameters class, and the IDoubleSource interface method *getDoubleValue*() takes care of reading the values of the regressor variables, the simulation of outcomes or probabilities based on regression models is straightforward:

```
      marriageScore = Parameters.getRegMarriageFit().
getScore(this, Person.Regressors.class);
      divorceProb = Parameters.getRegDivorce().
getProbability(this, Person.Regressors.class);
      workProb = Parameters.getRegInWork().
getProbability(this, Person.Regressors.class);
```

Again, if the specification of the model is changed by adding or removing covariates, or if new coefficient estimates become available, nothing has to be changed in the code, except for adding any new covariate to the Person.Regressors Enum and providing a method for the new case in the *getDoubleValue*() method.[23]

4.6.4 The Household Class

This class contains a list of all household members and is able to count the number of adults and children in the household. It is defined as an Entity class and is therefore linked to a table with the same name in the database. It implements the EventListener interface because it responds to calls by the scheduler requesting that the household composition is updated.

4.6.5 The PersonsCollector Class

The Collector collects statistics and manages the persistence of the simulation outputs on the database. It extends the AbstractSimulationCollectorManager class and requires, similarly to the Model, the implementation of a *buildObjects*() method and a *buildSchedule*() method.

The *buildObjects*() method creates several CrossSection objects, which collect specific values from each individual in the population (Box 4.13).

[23]The change in specification is instead achieved by updating the regression coefficient input files (e.g. *reg_inwork.xls*).

Box 4.13 The PersonsCollector.*buildObjects*() Method

```
@Override
public void buildObjects() {

final PersonsModel model =
(PersonsModel) getManager();

ageCS = new CrossSection.Integer(model.getPersons(),
Person.class, "age", false);

nonEmploymentCS = new CrossSection.Integer(
model.getPersons(), Person.class, "getNonEmployed",
true);

employmentCS = new CrossSection.Integer(
model.getPersons(), Person.class, "getEmployed",true);

retiredCS = new CrossSection.Integer(
model.getPersons(), Person.class, "getRetired",true);

inEducationCS = new CrossSection.Integer(
model.getPersons(), Person.class, "getStudent",true);

lowEducationCS = new CrossSection.Integer(model.
getPersons(), Person.class, "getLowEducation",true);

midEducationCS = new CrossSection.Integer(model.
getPersons(), Person.class, "getMidEducation",true);

highEducationCS = new CrossSection.Integer(model.
getPersons(), Person.class, "getHighEducation",true);

}
```

The Collector's schedule is made up of two processes only, which take place at every simulation period: the CrossSections are updated (Processes.*Update*), and the persons and households are persisted in the database (Processes.*DumpInfo*) (see Box 4.14).

Box 4.14 The PersonsCollector.*buildSchedule*() Method

```
@Override
public void buildSchedule() {

EventGroup collectorSchedule = new EventGroup();
```

(continued)

```
collectorSchedule.addEvent(this, Processes.Update);
collectorSchedule.addEvent(this, Processes.DumpInfo);

getEngine().getEventList().schedule(collectorSchedule, 0, 1);

}
```

The Collector also implements the EventListener interface, featuring the Enum Processes and *onEvent*() method (see Box 4.15).

Box 4.15 Implementation of the EventListener Interface in PersonsCollector

```
public enum Processes {
       Update,
       DumpInfo;
}

@Override
public void onEvent(Enum<?> type) {
       switch ((Processes) type) {
       case Update:
            ageCS.updateSource();
            nonEmploymentCS.updateSource();
            employmentCS.updateSource();
            retiredCS.updateSource();
            inEducationCS.updateSource();
            lowEducationCS.updateSource();
            midEducationCS.updateSource();
            highEducationCS.updateSource();
            break;

       case DumpInfo:
            try {
DatabaseUtils.snap(((PersonsModel) getManager()).
getPersons());

DatabaseUtils.snap(((PersonsModel) getManager()).
getHouseholds());

            } catch (Exception e) {
                  log.error(e.getMessage());
            }
            break;
       }
}
```

As we have seen in Sect. 4.6.2, implementing the EventListener interface is not necessary, as the class can be activated by the scheduler using Java reflection. However, grouping all the updating in one single *Update* process improves on clarity.[24]

Updating the CrossSection objects only involves simple instructions such as

```
ageCS.updateSource();
```

Similarly, dumping the simulation outputs is done by the *DumpInfo* process and only requires

```
DatabaseUtils.snap(((PersonsModel) getManager()).getPersons());
DatabaseUtils.snap(((PersonsModel) getManager()).getHouseholds());
```

4.6.6 The PersonsObserver Class

The PersonsObserver builds graphical objects that allow monitoring and inspection of the simulation outcome in real time. It extends the AbstractSimulationObserver-Manager interface and, similarly to the other simulation managers (the Model and the Collector), requires the implementation of a *buildObjects*() method and a *buildSchedule*() method.

The *buildObjects*() method creates three plots. The first one (*agePlotter*) depicts the evolution of the average age of the simulated population: it takes the *ageCS* CrossSection from the Collector, with information on the age of each individual, and computes its mean (by creating a MeanArrayFunction object). Similarly, the *workPlotter* plots the frequency of students, retired, other non-employed and employed individuals in the population, and the *eduPlotter* plots the share of individuals with each educational level (Box 4.16).

Box 4.16 The PersonsObserver.*buildObjects*() Method

```
@Override
public void buildObjects() {

final PersonsCollector collector = (PersonsCollector)
getCollectorManager();

agePlotter = new TimeSeriesSimulationPlotter
("Age", "age(years)");
```

(continued)

[24] Because updating is a common activity, it is also defined as a *CommonEventType* Enum in the JAS-mine *event* library (together with saving). Passing the scheduler this Enum does not require implementing the EventListener interface. An example of this approach is implemented in the Observer.

```
agePlotter.addSeries("avg",
      new MeanArrayFunction(collector.getAgeCS()));
GuiUtils.addWindow(agePlotter, 250, 50, 500, 500);

workPlotter = new TimeSeriesSimulationPlotter
("Work status", "proportion");
workPlotter.addSeries("employed",
      new MeanArrayFunction(collector.getEmploymentCS()));
workPlotter.addSeries("non-employed",
      new MeanArrayFunction(collector.getNonEmploymentCS()));
workPlotter.addSeries("retired",
      new MeanArrayFunction(collector.getRetiredCS()));
workPlotter.addSeries("students",
      new MeanArrayFunction(collector.getInEducationCS()));
GuiUtils.addWindow(workPlotter, 750, 50, 500, 500);

eduPlotter = new TimeSeriesSimulationPlotter("Education
level", "proportion");
eduPlotter.addSeries("low",
      new MeanArrayFunction(collector.getLowEducationCS()));
eduPlotter.addSeries("mid",
      new MeanArrayFunction(collector.getMidEducationCS()));
eduPlotter.addSeries("high",
      new MeanArrayFunction(collector.getHighEducationCS()));
GuiUtils.addWindow(eduPlotter, 1250, 50, 500, 500);

}
```

Other plots can be easily added. In particular, by building on the JFreeChart library, the CollectionBarSimulationPlotter class in JAS-mine allows to create histograms for representing distributions of given variables in the simulated population, at any given simulation period.

The schedule of the PersonsObserver class manages the updating of these three plots (Box 4.17). Here, the built-in JAS-mine Enum CommonEventType.*Update* is used, rather than a class-specific implementation of the EventListener interface as in the Collector. This requires scheduling the update of each graph separately, but allows for a better control of the display frequency. The latter is obtained by means of an extra parameter which is loaded into the GUI:

Box 4.17 The PersonsObserver.*buildSchedule*() Method

```
@GUIparameter
private Integer displayFrequency = 1;

@Override
public void buildSchedule() {
```

(continued)

```
getEngine().getEventList().schedule(new
       SingleTargetEvent(agePlotter,
       CommonEventType.Update), 0,
       displayFrequency);
getEngine().getEventList().schedule(new
       SingleTargetEvent(workPlotter,
       CommonEventType.Update), 0,
       displayFrequency);
getEngine().getEventList().schedule(new
       SingleTargetEvent(eduPlotter,
       CommonEventType.Update), 0,
       displayFrequency);

}
```

4.6.7 The StartPersons Class

The Start class initialises the JAS-mine simulation engine and defines the list of models to be used. In general, the Start class is designed to handle two types of experiments:

- performing a single run of the simulation in **interactive mode**, through the creation of a Model and related Collectors and Observers, with their GUIs;
- performing a single run of the simulation in **batch mode**, through the creation of the Model and possibly the Collectors; this involves managing parameter setup, model creation and execution directly, and is aimed at capturing only the simulation's numerical output;

The Start class is ignored when performing a **multi-run session** (whose structure is defined in a class extending the MultiRun interface) where the simulation is run repeatedly using different parameter values, so as to explore the space of solutions and produce sensitivity analyses on the specified parameters.

The Start class implements the ExperimentBuilder interface, which defines the *buildExperiment*() method. This method creates managers and adds them to the JAS-mine engine. In *Demo07*, the simulation is run in interactive mode (Box 4.18).

Box 4.18 The StartPerson Class

```
public class StartPersons implements
ExperimentBuilder {

public static void main(String[] args) {
```

(continued)

```
        boolean showGui = true;

        StartPersons experimentBuilder = new StartPersons();

        final SimulationEngine engine =
                              SimulationEngine.getInstance();
        MicrosimShell gui = null;
        if (showGui) {
                gui = new MicrosimShell(engine);
                gui.setVisible(true);
}

        engine.setExperimentBuilder(experimentBuilder);

        engine.setup();

}

@Override
public void buildExperiment(SimulationEngine engine)
{

PersonsModel model = new PersonsModel();
PersonsCollector collector = new PersonsCollector(model);
PersonsObserver observer = new
PersonsObserver(model, collector);

engine.addSimulationManager(model);
engine.addSimulationManager(collector);
engine.addSimulationManager(observer);

}

}
```

The Start class contains the standard *main*() method which allows a Java application to run. By selecting the "Run As Java Application" option from the Eclipse menu, this procedure launches the JAS-mine GUI, creates a model instance and gives it to the simulation engine. It then creates a Collector and an Observer and calls the *setup*() method of the simulation engine, which has the task of loading the experiment into memory.

The JAS-mine GUI contains a mask for setting the specific Model parameters, another mask for defining the specific Observer parameters and the three dynamic graphs defined in the Observer class. Figure 4.1 depicts the graphical output of one simulation run.

The Tools > 'Database explorer' tab in the JAS-mine GUI allows to browse the input and output databases. By selecting a specific database and pressing the 'Show database' button, the data can be explored in the default web browser using SQL

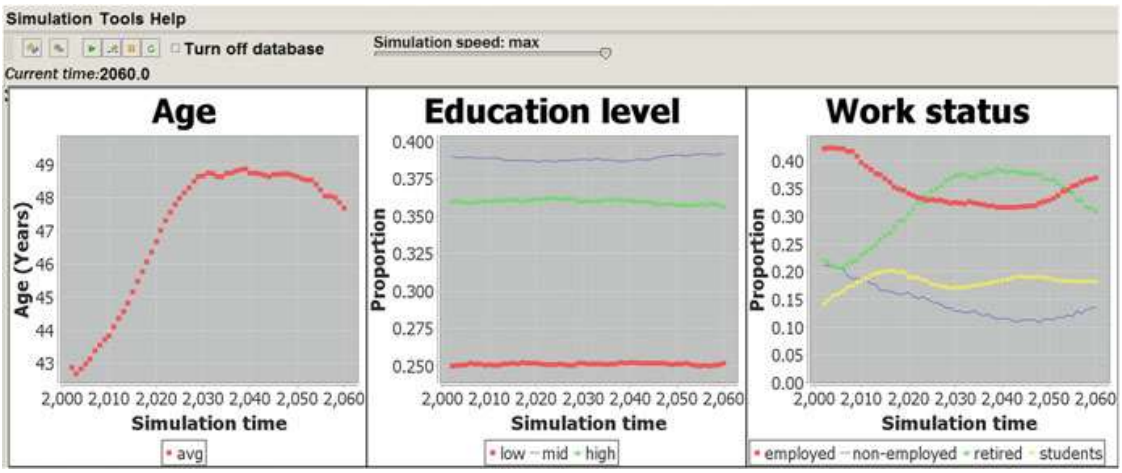

Fig. 4.1 The graphical output of one simulation run

commands. The output tables can also be exported in CSV format for subsequent analysis using specific statistical tools by typing:

```
CALL CSVWRITE('person.csv', 'SELECT * FROM PERSON ');
```

4.7 Conclusions

The JAS-mine simulation platform achieves a convergence between agent-based and microsimulation tools. Its main goal is to speed up model development, facilitate model documentation, and foster model testing and sharing. The rationale behind this choice lies in the observation that computer power is always increasing, while researchers' time is not. Also, large-scale simulation projects are generally beyond the reach of a single scientist. Even when they remain under the control of a restricted group of people, they generally require a prolonged effort, often on a stop-and-go basis. The possibility of building on previous work done by the same authors or by other researchers is crucial. Simulation modelling needs cooperative development. The choice of an entirely open-source tool should be evaluated in this light. Moreover, JAS-mine does not force the user to adopt predefined solutions to the problems faced in simulation modelling. By offering a set of libraries that extend the capability of the standard Java classes, JAS-mine leaves entirely open the possibility of using other libraries and tools, either as an alternative or on top of the JAS-mine toolkit.

Acknowledgments Matteo Richiardi has benefited from support by Piedmont Region (research project: "From work to health and back: The right to a healthy working life in a changing society") and Collegio Carlo Alberto (research project: "Causes, Processes and Consequences of Flexsecurity Reform in the EU: Lessons from Bismarkian Countries"), as well as from support by a Marie Curie Intra European Fellowship within the 7th European Community Framework Programme.

We are grateful to Michele Sonnessa, who developed an earlier version of the platform and has contributed to the coding of the microsimulation model.

References

Baekgaard, H. (2002). *Micro-macro linkage and the alignment of transition processes: Some issues, techniques and examples*. National Centre for Social and Economic Modelling (NATSEM) Technical paper No. 25.

Billari, F., & Prskawetz, A. (2003). *Agent-based computational demography*. Berlin: Springer.

De Menten, G., Dekkers, G., Bryon, G., Liègeois, P., & O'Donoghue, C. (2014). LIAM2: A new open source development tool for discrete-time dynamic microsimulation models. *Journal of Artificial Societies and Social Simulation, 17*(3), art. 9.

Gilbert, N., & Troitszsch, K. (2005). *Simulation for the social scientist* (2nd ed.). Maidenhead: Open University Press.

Keller, A.M., Agarwal, S., & Jensen, R. (1993). Enabling the integration of object applications with relational databases. *ACM SIGMOD Proceedings*.

Klevmarken, A. (2002). Statistical inference in micro-simulation models: Incorporating external information. *Mathematics and Computers in Simulation, 59*(1–3), 255–265.

Li, J., & O'Donoghue, C. (2012). Simulating histories within dynamic microsimulation model. *International Journal of Microsimulation, 5*(1), 52–76.

Li, J., & O'Donoghue, C. (2014). Evaluating binary alignment methods in microsimulation models. *Journal of Artificial Societies and Social Simulation, 17*(1), art. 15.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., & Balan, G. (2005). MASON: A multiagent simulation environment. *Simulation, 81*(7), 517–527.

Luna, F., & Stefansson, B. (2000). *Economic simulations in Swarm: Agent-based modelling and object oriented programming*. Dordrecht: Kluwer.

Mannion, O., Lay-Yee, R., Wrapson, W., Davis, P., & Pearson, J. (2012). JAMSIM: A microsimulation modelling policy tool. *Journal of Artificial Societies and Social Simulation, 15*(1), art. 8.

Minar, N., Burkhart, R., Langton, C., & Askenazi, M. (1996). The Swarm simulation system: A toolkit for building multi-agent simulations. *Santa Fe Institute Working Paper*, 96-06-042, Santa Fe.

North, M. J., Collier, N. T., Ozik, J., Tatara, E., Altaweel, M., Macal, C. M., Bragen, M., & Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling, 1*, 1–26.

Richiardi, M. (2012). Agent-based computational economics. A short introduction. *The Knowledge Engineering Review, 27*(2), 137–149.

Richiardi, M. (2013). The missing link: AB models and dynamic microsimulation. In S. Leitner & F. Wall (Eds.), *Artificial economics and self organization* (Lecture notes in economics and mathematical systems, Vol. 669). Berlin: Springer.

Statistics Canada (2009). *Modgen version 10.1.0 developer's guide*. http://www.statcan.gc.ca/eng/microsimulation/modgen/doc/devguide/dev. Accessed 17 Dec 2014.

Stephensen, P. (2014). An information-loss-minimizing approach to multinominal alignment in microsimulation models. *DREAM Working Paper*, 2014:4.

Wilensky, U. (1999). *NetLogo user manual*. https://ccl.northwestern.edu/netlogo/docs. Accessed 10 Apr 2015.