Working Paper no. **134**

JAS 2: a new Java platform for agent-based and microsimulation modeling.

Matteo G. Richiardi
University of Turin, Collegio Carlo Alberto and
LABORatorio R. Revelli, Turin

Michele Sonnessa
Zero11.it

August, 2013

JAS 2: a new Java platform for agent-based and microsimulation modeling.

Matteo Richiardi
University of Torino, Department of Economics and Statistics,
Campus Luigi Einaudi, Lungo Dora Siena 100A, 10153 Torino, Italy
Collegio Carlo Alberto and LABORatorio Revelli,
Via Real Collegio 30 - 10024 Moncalieri (Torino), Italy

Michele Sonnessa
Zero11.it

This version: August 2013

Abstract:
We present JAS 2, a new Java platform that aims at providing a unique simulation tool for discrete-event simulations, including agent-based and microsimulation models.

With the aim to develop large-scale, data-driven models, the main architectural choice of JAS 2 is to use whenever possible standard, open-source tools already available in the software development community. The main value added of the platform lies in the integration with RDBMS (relational database management) tools through ad-hoc microsimulation Java libraries. 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 (which is typical of a database) into the object-oriented simulation model, where each category of individuals or objects that populate the model is represented by a specific class, with its own properties and methods. JAS 2 allows to separate data representation and management, which is automatically taken care of by the simulation engine, from the implementation of processes and behavioral algorithms, which should be the primary concern of the modeler. This results in quicker, more robust and more transparent model building.

Acknowledgements:
This work has benefited from financial support from 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").

Keywords: simulation platform, microsimulation, agent-based, software, open-source.

JEL codes: C63, C88

1. Introduction

In this paper we present a new simulation platform for all types of discrete-event computer simulations, in particular agent-based and microsimulation models.

The tool, named JAS 2, consists of a Java library of functions and classes to speed up the development of simulation models, and is explicitly designed for handling *large-scale* applications. In addition to providing pre-set / ready-to-use functions, JAS 2 also provides a pattern that guides the user through program creation, allowing for maximum uniformity and transparency in the model structure. From a methodological viewpoint, JAS 2 extends the *Model-Observer* paradigm introduced by the Swarm experience (Minar *et al. 19*96) and introduces a new layer in simulation modeling, 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 compute the statistics both for use 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.

Flexibility in model design and implementation, *scalability* of the code (i.e. the code complexity must increase approximately linearly with the complexity of the underlying model, and must remain highly readable for debugging, cooperative development, and documentation), *efficiency* in data exchange and input-output communication, and *compatibility* with external software solutions and tools were the main specifications for the project. Consequently, we made the following architectural choices in the development of JAS 2: first, we opted for an object-oriented programming language (OOP), which provides a natural and intuitive way of modeling populations of agents; second, we strictly adhered to the open-source paradigm; third, rather than developing our own ad-hoc grammar and syntax –as the popular Netlogo environment for agent-based modeling (Wilenski 1999) and Liam2 toolkit for microsimulation modeling (De Menten *et al.* 2012) for instance– we assembled a set of open tools to "manufacture" a simulation model, making use whenever possible of solutions already available in the software development community.

These decisions entail a longer learning curve for the user than a dedicated, language-specific software package, but allow for a more professional approach to simulation modeling compared to specific proprietary environments. There are two main advantages of using an open architecture where external libraries can be added and utilized: one is the possibility to easily integrate external functions that can be added as plug-ins; the other is the possibility to freely extend and modify the functions available on the platform, possibly within a cooperative effort of the community of users. Moreover, an open source application makes it easier to share and test the models.

From a computational perspective, any simulation model (including microsimulations and agent-based models) can be viewed as a black box receiving input data of varying complexity and producing, for each simulation period, outputs of varying complexity. The microstructure nature of both microsimulations and agent-based models impinges on this complexity. In particular, it requires a richer data structure to keep track of the evolution of the system, possibly consisting of different archives in relation with each other (Figure 1).

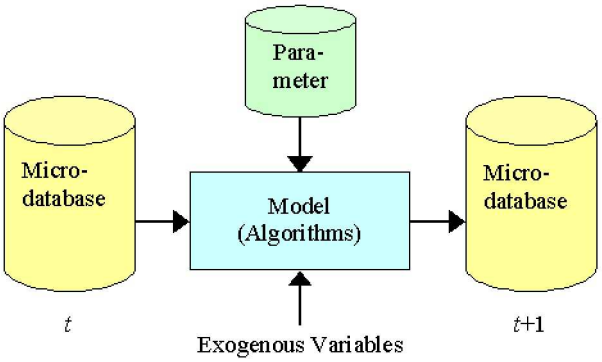

Figure 1. Structure of a simulation model

Clearly, data management is a major factor to be weighed in for the creation of a simulation tool. Building on the vast number of software solutions available, JAS 2 allows the user to separate data representation and management from the implementation of processes and behavioral algorithms. The main value added of the platform lies in the integration with relational database management systems (RDBMS) through ad-hoc Java libraries. This distinguishes JAS 2 from other open-source Java-based simulation platforms as MASON (Luke *et al.* 2005) and Repast (North *et al.* 2013). The management of input data persistence layers and simulation results in JAS 2 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. This also allows to separate data creation from data analysis, which is crucial for understanding the behavior of the simulation model. 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 on 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 RDBMS make it feasible to keep track of a much larger set of variables, for later analysis. Also, the statistical techniques envisaged, and the specific modeler'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 the brevity, transparency and clarity of the code.

The paper is organized as follows: section 2 discusses the analogies between dynamic microsimulations and agent-based models; section 3 describes the main architectural choices in the development of JAS 2; section 4 presents the structure of a simulation project using JAS 2; section 5 exemplifies with a simple demographic application; section 6 offers our concluding remarks.

2. Agent-based models and microsimulations as computer models

JAS 2 provides a unique tool for both microsimulations and large-scale agent-based applications. The trend toward a convergence in the two modeling approaches is analyzed in [Richiardi 2013].

Dynamic microsimulations are computer models aimed at applying specific processes (as aging, educational choices, labor market events, household formation, etc.) to a representative sample of a given population, in order to forecast its evolution or understand the distributional impact of some policy of interest. While some processes might be deterministic (as aging, or pension eligibility), other processes are stochastic (as mortality, or retirement choices) and are generally estimated in the data.

Agent-based models also take an initial population and simulate it forward in time. The main differences between the two approaches can be traced down to the following: (i), microsimulations are more policy-oriented, while agent-based models are more theory oriented; (ii) microsimulations generally rely on a partial equilibrium approach, while agent-based models are most often closed models. The initial population in an agent-based model is typically not meant to reproduce a real population of agents: for example, in a labor market model with firm creation all individuals might be initiated as unemployed, or randomly employed. The focus is on the emergence of aggregate patterns from the interaction of the individual agents, with the aim to replicate some observed stylized fact (business cycle fluctuations, for instance). Accordingly, the value of the parameters that drive the processes are chosen ad-hoc, or only roughly calibrated with real data. However, in their struggle to replace dynamic stochastic general equilibrium (DSGE) models, agent-based models are becoming more empirically oriented. At the same time, microsimulations are becoming more complex, by including more behavioral responses and general equilibrium feedbacks.

If the two approaches retain different goals and perspectives, from a mathematical and computational perspective they are identical. Both agent-based models and microsimulations 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. 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.

With some confusion in the notation, computer simulations can be organized in two categories: continuous and discrete. *Continuous simulations* break up time into regular time slices (Δt) and 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 best thought to progress linearly between one point in time and the next. By contrast, *discrete simulations* are characterized by irregular timeframes that are punctuated by the occurrence of discrete events. 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 be discretized.

The *event list* order the events and the simulation is performed by extracting the event that is closest in time and submitting it to the model's agents, which change their state according to the signal (corresponding to the event) they have received. In the case of continuous simulations, the order of the processes that are applied must be exogenously assumed (and the assumption must be coherent with the specification of the model used for estimating the coefficients

governing each process). The events may be generated and scheduled also while running the simulation and not only in the initial planning phase.

3. The JAS 2 library

A software tool with the specifications highlighted in section 1 poses a number of IT challenges:

i. the need to interface with data storage systems;
ii. the definition of standards for most common data structure representations, so as to minimize errors and increase model readability;
iii. the need to store simulation outputs as time series of population states over time;
iv. a flexible definition of the algorithms that govern the dynamic processes.

The data structures most used in simulation modeling can be organized in two macro categories: an *input database* for running the simulation, and an *output database* for analyzing it. The input database contains a) sets of parameters and coefficients and b) the initial population to be simulated forward in time. The output database records the changes in the simulated population, either by sampling it at different points in time (continuous simulation) or by recording the timing of the events which happen to each agent in the population, including agent creation (discrete simulation). The output database historicizes the state of the system, including the initial period. For completeness, it should also contain a copy of the parameters and coefficients used in the simulation, so as to avoid indeterminacies about how the data were produced. Therefore, the output database contains (a copy of) the input database, and completely describes a simulation run.

3.1 The input database

The input database contains one table for each entity (agent type), populated with the initial samples, plus tables with coefficients and parameters. The entity tables are characterized by a unique or *primary key* and a set of characteristics that can be expressed as numerical or categorical values. Relations with other entities are represented by *foreign keys*, or fields that refer to the primary key in the table containing the description of the related entity.

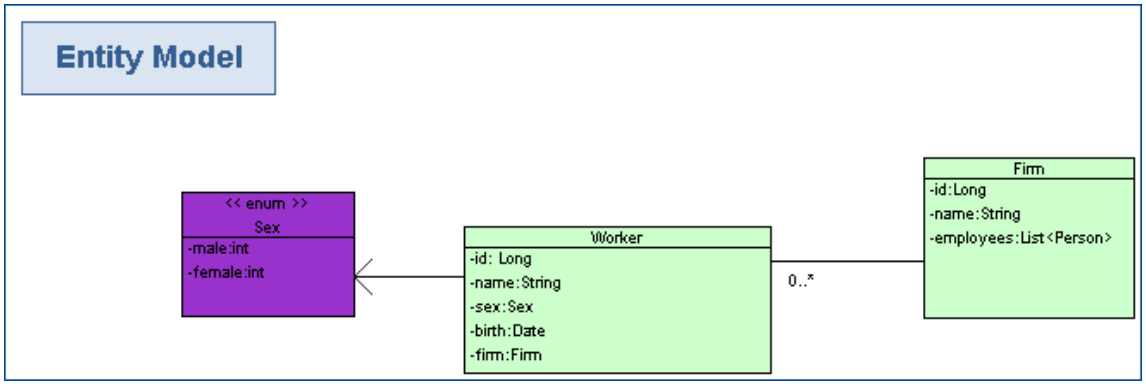

Figure 2. Example of entity tables

The example in Figure 2 describes two related entities: firms and workers. Representation in the database requires the unique identification of each worker and each firm as well as a

foreign key in the *workers* table containing the identification of the firm for which the individual is currently working. This relieves the object-oriented simulation model from the need to keep explicit memory of such relationship (or of any other relationship, e.g. firms for which workers have worked in the past), by including a specific pointer among the class properties. Of course pointers might well be used, and the relationship uploaded in the memory for faster access, but JAS 2 offers the option to trade off execution speed for code brevity and clarity.

The set of parameters and coefficients contains the values that make up the simulation scenario, specifying the number of individuals and time periods to be simulated, the processes to be applied, the alignment and matching methods to be used if relevant, and the estimated coefficients of the regression models. The parameters may be punctual, in which case they are represented by object dictionaries expressed in the form of key-value maps: values can be obtained by querying the dictionary by key. Specific keys can be used, for example to define which model to apply to a certain process (linear regression, logit, probit, etc). Storage of more complex data structures, like regression coefficients, can be represented using specific one-line tables. Field names refer to the name of the covariates, while the values of the estimated coefficients are contained in the row field. When using several equations different tables or rows must be created, as is the case with multinomial logit/probit. Structures may be even more complex and associate sets of keys to sets of values (as when a regression is estimated separately for men/women). In this case one or more values can be obtained by querying the dictionary using a set of keys. Another example of complex coefficient structure is demographic forecasts coming from outside the model, which are used to align the evolution of the simulated population. The keys in this case are time, gender, age group, immigration status, etc., and the associated coefficients represent the forecasted size of the population in each cell.

3.2 The output database

The output database contains a (unique) identifier of the simulation run, with the associated parameters[1], as well as a "snapshot" of the population (or of individual agents, in the case of discrete simulations) at various points in time.

Before the start of the simulation, JAS 2 configures the output database, copying all the tables contained in the input database, and adding an experiment (simulation run) key and a time key to all relevant tables. When running the simulation JAS 2 creates a unique experiment ID, performs, when prompted, snapshots of individual objects or collections of objects, connects to the output database and saves them as new records in the appropriate tables, with the run ID as experiment key and the current simulation period as time key.

3.3 The ORM paradigm

The software paradigm that is best suited to represent and manipulate population data is object-oriented programming (OOP), as described extensively in Luna *et al.* (2000) and Gilbert *et al*. (2000). On the other hand, input and output data (especially in complex projects) are best stored in a relational database. Unfortunately, database relational modelling is less intuitive than OOP and requires a specific language (SQL) to retrieve and modify data.

[1] Including parameters selected directly by the user using JAS 2 graphic interface.

In JAS 2, the interaction between the simulation and the (input and output) data is inspired to Object-Relational Mapping (ORM), a programming approach that facilitates the integration of object-oriented software systems with relational databases (Keller, 1993). An ORM product uses an object-oriented interface to provide services on data persistence, while abstracting at the same time from the implementation characteristics of the specific RDBMS (database management software) used. The main advantages of using an ORM system are:

i. the masking of the implementation of the relational model in an object-oriented model;
ii. high portability compared to the DBMS technology adopted: no need to rewrite data input queries on database when changing DBMS, simply modify a few lines in the configuration of the ORM used;
iii. a drastic reduction in the amount of code to be written; the ORM masks behind simple commands the complex activities of data creation, extraction, update and deletion. These activities take up a considerable percentage of the time required for writing, testing and maintenance. Moreover they are inherently repetitive, thus increasing the chance of errors when writing the implementation code.

The most common ORM products available today[2] offer a number of functions that would otherwise be performed manually by the programmer; in particular, the operations of loading the object graph based on association links defined at language level, and reading/writing/deleting are entirely automated. For instance, loading an instance of the Student class may result in the automatic loading of data concerning the student's exam grades.

The use of an ORM facilitates the achievement of higher quality software standards, improving in particular its correctness, maintainability, potential evolutions and portability. On the down side, choosing an ORM paradigm introduces a software layer that impacts on performance, an aspect that is relevant to data-intensive applications like simulations. Translating the entity-relational model that is typical of a database into an object-based model requires additional activities that may slow down data upload and reading. Given the continuous increases in the speed and power of modern computers, we opted for a lean architectural structure even at the cost of slowing down the simulation engine.

3.4 Reading and writing

The representation of the sample population is fully adherent to the standards used in IT systems to store entities and relations between entities. Consequently, population modeling can be performed according to standard strategies for modeling object classes and their persistence on database. In particular, ORM requires relationships between objects to be implicitly modeled. The ORM engine translates these relationships into foreign keys in the relational model.

[2] JAS 2 uses Hibernate (http://www.hibernate.org/).

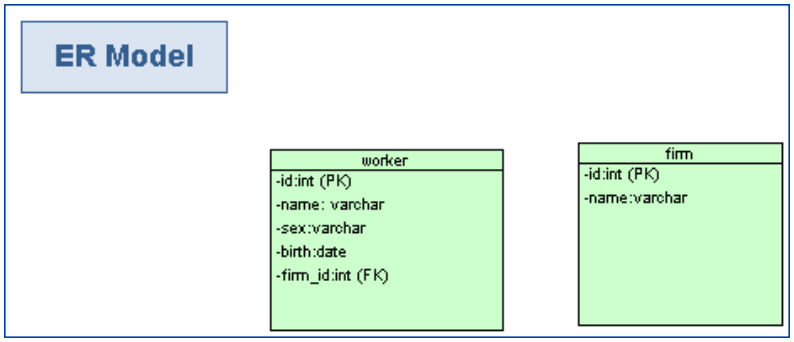

Figure 3. Example of an Entity/Relationship (E/R) diagram

For instance, in the Entity/Relationship (E/R) diagram of Figure 3, only the *firm_id* field containing the primary key of the firm table requires specification. In case all records concerning workers related to one firm need to be obtained, without using ORM a specific SQL query should be created, then run, to extract and insert data in an object intended to represent the connected entity. When using ORM the persistence engine is simply requested to get object of the *Firm* class corresponding to the desired identifier and the object's relational graph is loaded automatically, including related workers (objects of the *Worker* class). For example using the notation *worker.getFirm().getName()* will read from the database the name of the company where a worker is employed without the need of defining any SQL query, not differently from what one would do for reading the same information from the *Firm* object itself, accessed through a specific pointer in the *Worker* class.

3.4.1 Annotations

Since Java 5 annotations were introduced to represent attributes/adjectives assigned to specific parts of code as classes or properties. Annotations decorate the elements they are associated to, in the sense that they attribute meanings that can be used to add collateral information to objects.

Annotations make the definition and the use of coefficients tables more powerful and flexible. For example, a table is created to represent and manage the mapping of two characteristics –minimum retirement age and expected residual lifetime– for each sex-age group of the simulated population. The table contains four fields: *age*, *sex*, *retirementAge* and *residualLifeTime*. These fields have in fact different semantics: the first two correspond to research keys in a key-value dictionary, while the last two represent specific values.

ORM allows the construction of a Java class, for example called *CoefficientA*, that contains the four properties corresponding to the table fields; their values can then be read by the ORM engine. In order to populate the dictionary automatically the properties of the *CoefficientA* class can be "decorated" using the JAS 2 ad-hoc *CoefficientMapping* annotation.

```
@Entity
@CoefficientMapping(keys={"age", "sex"},values={"retirementAge", "residualLifetime"})
public class CoefficientA {
private Integer age;
private Sex sex;
private Integer retirementAge;
private Double residualLifeTime;
        …
```

```
}
```

The *Entity* annotation informs the ORM engine that the *CoefficientA* class corresponds to a table in the database which bears the same name as the class and contains the fields corresponding to the object's properties.[3] A JAS 2 library will then request the ORM engine to read the data contained in the table and to include them in a key-value structure that can be easily queried using an instruction like the following:

```
MultiKeyCoefficientMap coefficientA = DatabaseUtils.loadCoefficientMap(CoefficientA.class);
int retirementAge = coefficientA.get(30, Sex.Female, "retirementAge");
double residualLifetime= coefficientA.get(30, Sex.Female, "residualLifetime");
```

where the first two parameters of the *get* function are the two keys and the last two (*retirementAge*, *residualLifetime*) represent the name of the value variable.

This method for accessing parameter tables may appear convoluted and cumbersome. The same result can be achieved more rapidly by placing the map values in an excel sheet (Figure 4).

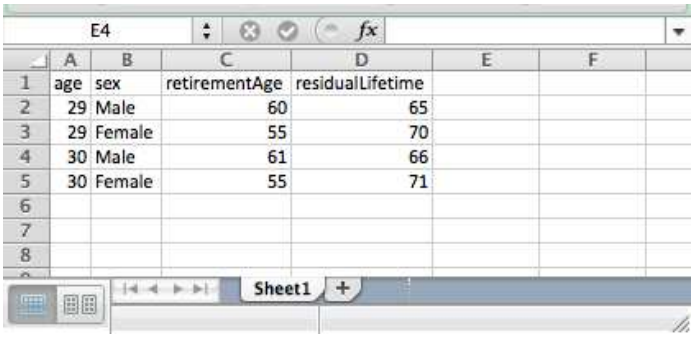

Figure 4: Example of an excel sheet containing parameters

The parameters are then loaded using a specific JAS 2 interface:

```
MultiKeyCoefficientMap coefficientA = ExcelAssistant.loadCoefficientMap("input/coeffA.xls", "Sheet1", 2, 2);
```

Only the number of key columns and "value" columns need to be specified. Clearly this process is much easier but it does not allow for significant parameter typification (since Excel is not as rigid as a database). Moreover, it is more error prone as accidental modifications to the Excel sheet might lead to incorrect parameter loading.

3.4.2 Historicizing agents

In a microsimulation the state of the population is historicized at each simulated point in time in order to get a time series of states for each entity. This is against the principles of ORM which implements the persistence of the object graph on the database as last saved and updated. Historicizing the state of an object in time is not explicitly envisaged in an ORM system; the tool presented here provides an out-of-the-box solution to handle this need.

[3] Through specific parameters names and mapping types between tables/fields and the corresponding Java classes/properties can be changed (see below).

JAS 2 requires that all tables to be historicized have a primary key consisting of the object ID (individual, family, firm, etc.), the ID of the simulation experiment and the simulated time. Those three keys will allow the JAS 2 persistence engine to provide values for the relevant fields. It is sufficient to ask JAS 2 to save an object or a collection of objects; the system adds the run ID and the simulation time and creates new records in the appropriate tables.

The primary key of a class to be historicized should be declared, in the simulation code, as follows:

```
@Entity
public class Agent {
   @EmbeddedId
   private PanelEntityKey id;
   …
}
```

The *@EmbeddedId* annotation is a way Hibernate use to tell the Java interpreter that the primary key is represented by a complex object (rather than a simple value). The *PanelEntityKey* object is a composite key and implies the declaration (in the database) of the three fields required for historicizing: *id*, *simulationTime* and *simulationRun*.[4]

3.5 The Model-Collector-Observer paradigm

The Swarm protocol for agent-based platforms architecture (Minar *et al. 19*96) recommend splitting the simulation into an *internal* Model and an *external* Observer. These two aspects of the artificial world should remain markedly separate.

The purpose of the Observer is inspecting the model's objects. Through the Observer the state of simulation can be monitored and graphically represented in real time, while the simulation is running. However, for the purpose of analysis and validation, the Observer alone may not be adequate, because it implies the need to define in advance the aggregations on which to analyze the simulation outcome. A variation in perspective requires re-running the experiment.

According to a different approach, the simulation is aimed exclusively at producing numerical outputs which can be analyzed in depth ex-post using ad-hoc statistical-econometric tools.

JAS 2 combines these two different approaches extending the Model-Observer paradigm so as to include an intermediate structure that calculates statistical values and persists simulation

[4] More in details, the PanelEntityKey class is defined in JAS 2 as follows:

```
@Embeddable
public class PanelEntityKey implements Serializable {
   @Column(name="id")
   private Long id;
   @Column(name="simulation_time")
   private Long simulationTime;
   @Column(name="simulation_run")
   private Long simulationRun
   ...
}
```

but the user does not need to know how this class look like.

modeling outputs in the database in the most transparent way, minimizing the impact on model implementation.

In the JAS 2 architecture agents are organized and managed by components called *managers*. 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 graphic widgets objects to inform in real time on the state of simulation and to define the frequency of updates of these objects.

JAS 2 allows multiple Models (and multiple Collectors and Observers) to run simultaneously, since they share the same scheduler (known as a *singleton*). 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*[5] 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.

4. The structure of a simulation project

4.1 The package structure

A microsimulation/agent-based project using standard Java annotation is organized according to the following package structure:

- **data:** a package containing the classes that describe the structure of coefficient, parameter and agent population tables contained in the database to be loaded by the ORM. When using Excel files to specify input data no specific classes need to be included in this package;
- **model:** a 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**: a subpackage containing the definition of the enumerations used (if any);[6]
- **experiment**: a 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)

[5] The *Model* classes can extend the *AbstractSimulationManager* abstract class which provides a standard implementation of the getId() and engine getter and setter methods.

[6] Enumerations specify a set of predefined values that a property can assume. These values might be categorical (strings, e.g. sex), 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*. Through enumerations the ORM automatically manages reading/writing operations in both text and numerical format.

are defined, the *MultiRun* class that manages repeated runs for parameter exploration, and the *Observer* manager class for online statistics collection and display.

- **algorithms**: a package containing classes that implement algorithms for determining events and applying processes to the agents (for example, econometric models, alignment and matching methods, etc.). These implementations, in a cooperative effort of users, are potential candidates to extend the set of standard functions included in the JAS 2 libraries;
- **utils**: a package containing utility classes for technical support to implementation.

In addition to sources, the project also contains two folders for data input-output. The input folder contains data on database inputs in excel or H2 embedded formats. The output folder contains the output of different simulation experiments. At the beginning of each run, JAS 2 creates a sub-folder that is labeled automatically[7] 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.

By default, JAS 2 executes the simulations in embedded mode: the databases are modified directly by the JDBC driver included in JAS 2.The standard database uses a H2 database engine. Other databases supporting embedding can be used, like Microsoft Access, Hypersonic SQL, Apache Derby, etc. To change the database type, it is sufficient to reconfigure the file *persistence.xml*, which otherwise does not need to be modified. Also, by pointing the file *persistence.xml* to a database server it is possible to use the database in server mode, through a network interface.

4.2 The Model and the schedule

A Model should implement the *SimulationModel* interface or extend the *AbstractSimulationModel* class. Two methods are required in the Model class. The *buildObjects* method should contain the instructions to create all the agents and the objects that represent the virtual environment for model execution. The *buildSchedule* method should contain the planning of the events for the simulation.

In the following example the first instruction loads a list of agents, instances of the *Agent* class, from an input database table using the JAS 2 integrated ORM system:

```
public void buildObjects() {
agentsList = (List<Agent>) DatabaseUtils.loadTable(Agent.class);
}

public void buildSchedule() {
EventGroup s = new EventGroup();
    s.addEvent(this, Processes.AlignPopulation);
    s.addCollectionEvent(agentsList, Agent.Processes.Age);
    getEngine().getEventList().schedule(s, 0, 1);
    getEngine().getEventList().schedule(new SingleTargetEvent(this, Processes.Stop), 2);
}
```

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

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

can be specified in case of recurring events characterized by a specific frequency. An event can be created for a specific recipient. In particular, an event can be created and managed by the simulation engine (a system event, e.g. simulation stops), 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 the example above a group of events (labeled *s*) is created to be run for the first time at time 0 and then repeated at each simulation period. Within an event group, events are run sequentially, as specified in the code. The first event in the event group *s* is targeted to the model itself and entails running the *AlignPopulation* method, which performs alignment of the simulated population to outside forecasts. The second event (*Age*) is sent to all the individuals in the simulated population and entails aging. Finally, the end of the simulation is scheduled at time *t=2* and will be notified to the model itself.

A class can receive and process events after implementing the *EventListener* interface and defining the *onEvent* method that will receive specific enumerations to be interpreted.

In the example, the model defines an enum called *Processes* as follows:

```
public enum Processes {
    AlignPopulation,
    Stop;
}
```

The *onEvent* method decodes this object and performs the required action:

```
public void onEvent(Enum<?> type) {
    switch ((Processes) type) {
    case AlignPopulation:
            […]
            break;
    case Stop:
            getEngine().pause();
            break;
    }
}
```

Analogously, the Person class also defines an enum called *Processes*, which contains only the *Age* case.

Note that events can be scheduled dynamically and need not be planned in advance when constructing the model. For instance, events can be added by the agents themselves, based on their behavioral rules. This simply requires accessing the event list through a singleton instance of the simulation engine, with the following instruction:

```
SimulationEngine.instance.getEventList()
```

4.3 Additional model parameters

Managers (typically, the Model) can define some variables that are regarded as simulation parameters, in addition to those defined in the input database. To this end all the properties of the manager class that are marked with the *@ModelParameter* annotation are regarded as model parameters.

Accordingly, the JAS 2 engine generates a window for each defined manager, with the list of variables defined as parameters, so that the user can change them via the GUI before constructing and running the model. These parameters are added to the ones defined in the input database tables, for example to define specific simulation scenarios on the fly, in interactive mode. The persistence system integrated in JAS 2 records these values in a specific table of the output database, together with the date and time of execution of the simulation run.

4.4 The agents

As we have seen, the model, like all the classes that define the agents that populate the simulation, generally implements the *EventListener* interface which requires the definition of a *callback* method (*onEvent*). By implementing this interface an agent or any other class can respond to specific events that are planned in the JAS 2 scheduler. The events must be defined within the class itself using a specific *enum*.

This is the only technical element required to define an agent. From a modeling point of view an agent is characterized by the fact that it represents an entity of the simulated system and that it possesses state variables and behavioral processes/algorithms which can modify such variables and interact with other system elements.

The semantic difference between an agent and a simple class, specified according to the OOP paradigm, is therefore merely logical. All the elements required to define an agent are present in a generic class: properties and methods.

4.5 The *Start* class

The *Start* class serves to initialize and run the JAS 2 simulation engine and to define the list of models to be used. The Start class is designed to handle three types of situations:

- 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;
- 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 must implement the *ExperimentBuilder* interface, which defines the *buildExperiment* method. This method should create managers and add them to the JAS 2 engine. In the example below, a model called *DemoModel* is created and run in interactive mode:[8]

```
public static void main(String[] args) {
    boolean showGui = true;

    SimulationEngine engine = SimulationEngine.getInstance();
    MicrosimShell gui = null;
```

[8] 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.

```
if (showGui) {
        gui = new MicrosimShell(engine);
        gui.setVisible(true);
}
engine.setBuilderClass(StartDemo.class);
engine.setup();
}

@Override
public void buildExperiment(SimulationEngine engine) {
        DemoModel model = new DemoModel();
        PersonsCollector collector = new PersonsCollector(model);
        PersonsObserver observer = new PersonsObserver(model, collector);

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

5. An example: a simple demographic microsimulation

In order to sum up the technical aspects presented in the previous section we now provide a practical example of microsimulation modeling using JAS 2. The example closely follows the *demo03* sample model included in the Liam2 simulation platform.[9] The model describes a very basic population which is evolved in time through the application of a birth, aging and death process. The microsimulation is set up in continuous time, and each individual is historicized at constant time intervals (equivalent to a simulated year) in the output database.

The project structure is described in Figure 5 (the project is contained in a package named *it.zero11.microsim*).

[9] The code for the sample model can be downloaded at the JAS 2 website http://www.jasimulation.org. JAS 2 provides a plug-in for the open source Eclipse development environment (http://www.eclipse.org) to simplify model setup. The plugin installation instructions are also available at the JAS 2 website.

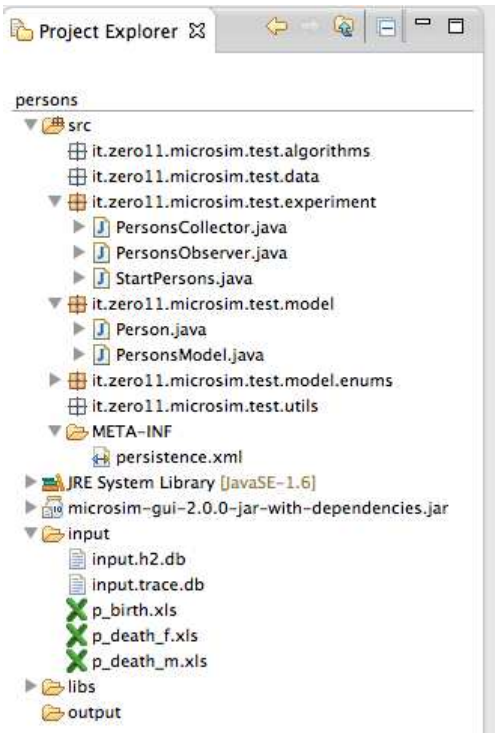

Figure 5. The *Persons* model structure

5.1 The *Person* class

Agents in the sample application are called persons. Their properties and methods are described in the *Person* class, included in the *model* package:

```
@Entity
public class Person implements EventListener {
    […]
     @EmbeddedId
    private PanelEntityKey id;

     private Integer age;

     @Enumerated(EnumType.ORDINAL)
     private Gender gender;

     @Column(name = "MOTHER_ID")
     private Long motherId;

    @Transient
     private Long ageClass;
     […]
 }
```

The class is decorated with a significant number of annotations that serve to describe to the ORM system how to connect the class properties to the database structure. The class is declared with the annotation *@Entity* that indicates to the system that this class corresponds to a table in

the database.[10] The use of a *PanelEntityKey* composite key indicates that this entity can be historicized by JAS 2. In other words, the output database will record the state of the objects belonging to this class at various points in time during the simulation. For some class fields the annotation *@Column* has been added to indicate the actual name of the field in the database corresponding to the object property. In case the two names coincide, there is no need to add the annotation. All the properties of the entity class that are not recorded in the database and that do not have a correspondent in the table must be marked with the annotation *@Transient*. Consequently the ORM system will ignore that property when reading from/writing on the database. Moreover, the properties characterized by enumeration are marked with the annotation *@Enumerated(EnumType.ORDINAL)* which indicates that the database will contain a numerical index to represent the value.[11] The mapping between indexes and values is determined by the order in which the items are specified in the *Enum* type definition (i.e. the order in which they are enumerated).

Enumerations are defined in devoted classes located in the *enums* subpackage. Gender is enumerated as follows:

```
public enum Gender {
   Male,
   Female;
}
```

Accordingly, the related database field will contain "0" for Male and "1" for Female.

The corresponding *Person* class table created in the H2 database is structured as follows:

COLUMN_NAME	TYPE_NAME	IS_NULL
ID	INTEGER	NO
AGE	INTEGER	YES
GENDER	INTEGER	YES
MOTHER_ID	INTEGER	YES
SIMULATION_TIME	INTEGER	NO
SIMULATION_RUN	INTEGER	NO

Table 1. Structure of the *Person* table.

As already pointed out, the simulator can be easily linked to any source of relational data, thanks to the ORM layer, by modifying the file *persistence.xml*.

The *Person* class also defines the methods owned by the agents. In particular, the class defines through enumeration the possible events (corresponding to processes) an agent can react to, through the *onEvent* method:

```
public enum Processes {
    Age,
```

[10] The name of the table is expected to correspond to the name of the class (in lower case). If the table has a different name it can be specified through the *@Table* annotation.

[11] To register the alphanumerical string representing the enumeration directly, the *EnumType.STRING* type must be specified.

```
    GiveBirth,
    Death;
}

public void onEvent(Enum<?> type) {
    switch ((Processes) type) {
    case Age:
            age += 1;
            break;
    case GiveBirth:
             if (this.gender.equals(Gender.Female) && this.age >= 15 && this.age <= 50) giveBirth();
            break;
    case Death:
            death();
            break;
    }
}
```

The birth function is applied only to women aged between 15 and 50. Aging simply implies an increase in the age variable. The birth process is defined in the following function:

```
public void giveBirth() {
    double pBirth = ((Number) model.getpBirth().getValue(this.age, model.getStartYear() +
            SimulationEngine.instance.getTime())).doubleValue();

    if (SimulationEngine.getRnd().nextDouble() < pBirth) {
        Person person = new Person();
        person.setAge(0);
        person.setMotherId(this.getId().getId());
        person.setGender( (SimulationEngine.getRnd().nextDouble() > 0.49 ? Gender.Male : Gender.Female) );
        model.getPersons().add(person);
    }
}
```

The agent asks the model which is her probability of having an offspring (see subsection below), given her age and year of birth (or calendar year).[12] A random number between 0 and 1 is extracted; if that number is lower than the probability of birth a new *Person* object is created, initialized and included in the list of agents the form the population.

Analogously, the death process is defined by the following method:

```
public void death() {
    MultiKeyCoefficientMap map = ( gender.equals(Gender.Male) ? model.getpDeathM() : model.getpDeathF() ) ;

    double pDeath = ((Number) map.getValue(this.age, model.getStartYear() +
            SimulationEngine.instance.getTime())).doubleValue();

    if (SimulationEngine.getRnd().nextDouble() < pDeath) model.getPersons().remove(this);
}
```

[12] No twins are allowed.

The probability of death depends on gender, in addition to age and time: therefore, two separate parameter files exist; the method selects the appropriate coefficient map.[13] In case death occurs, the agent is removed from the population.

5.2 The coefficients

Birth coefficients differ by age and year; they are listed in an Excel spreadsheet. Analogously, death coefficients are listed in two separate Excel files (one for men, the other for women).

When constructing the model (see subsection below) the tables are loaded into memory as a *MultiKeyCoefficientMap* object (see section 3.4 above).

5.3 The *PersonsModel* class

When the user presses the "*build model*" button on the graphical interface the JAS 2 engine invokes the *buildObjects* and *buildSchedule* methods of all defined managers: Model(s), Collector(s) and Observer(s). In particular, the Model is the object that coordinates model execution. It creates the other objects required to run an experiment and defines the order of events that will determine the evolution of the model. In the *Persons* model, the *buildObjects* reads as follows:

```
public void buildObjects() {
    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);
    persons = (List<Person>) DatabaseUtils.loadTable(Person.class);
}
```

Through the *ExcelAssistant* class, the JAS 2 engine is instructed to read from the Excel files *p_birth.xls*, *p_death_m.xls*, and *p_death_f.xls*, all located in the sub-folder *input*, which contain a sheet named *Sheet1*. The coefficient map is characterized by a key column (third parameter of the function) and 59 value columns, each representing a calendar year.[14]

The coefficient maps are queried (by the agents, see above) using age to identify the table row and the specific time index to identify the appropriate column.

Finally, the method loads the initial population from the input database into a list of agents (*persons*). The *loadTable* function only requires as argument the reference class: it automatically queries the input database and, through the ORM engine, loads the data from the corresponding table and creates the related objects.

After loading the data and building all the objects, the model class specifies the schedule of the simulation, with the method *buildSchedule*:

[13] Note that the parameters concerning the probability of death by gender and age could also have been stored in a unique Excel file, in different sheets, or even in a unique sheet, with gender as an additional key column. Which solution is to be preferred depends only on the modeler's preferences.

[14] As we have already discussed, an alternative method for imputing the coefficients is to store them as tables in the input database, define related annotated objects and load the data through the ORM engine (see again section 3.4).

```
public void buildSchedule() {
    EventGroup s = new EventGroup();
    s.addCollectionEvent(persons, Person.Processes.Age);
    s.addCollectionEvent(persons, Person.Processes.GiveBirth, false);
    s.addCollectionEvent(persons, Person.Processes.Death, false);
    getEngine().getEventList().schedule(s, 0, 1);
    getEngine().getEventList().schedule(new SingleTargetEvent(this, Processes.Stop), 2);
}
```

The *Age*, *GiveBirth* and *Death* events are grouped together and scheduled so that they are run at each step of the simulation (parameters 0, 1 indicate that the group is run at step 0 and then at each subsequent step). The group sends a message containing the sequence of events to each individual of the population.

Finally, the schedule plans the interruption of the execution at step 2: this ends the simulation.[15]

5.3 The *PersonsCollector* class

The manager acting as Collector has the same structure as the Model. It also implements the *buildObjects* and *buildSchedule* methods to define objects and execution times.

In particular it manages the *DumpInfo* event that requires the model to execute the following instruction:

```
public void dumpInfo() {
    DatabaseUtils.snap(((PersonsModel) getManager()).getPersons());
}
```

This instruction tells JAS 2 to make a snapshot of the population in the output database. Each record is saved with simulation time and experiment ID as primary keys, in addition to the agent ID. In this way the output table can contain data on the same entity at different points in time.

The Collector also defines a *CrossSection* object that produces a vector of values calculated by inspecting all the individuals in the population and manipulating one specific variable (age, in this example).[16]

The cross-section is defined in the *buildObjects* method of the Collector:

```
public void buildObjects() {
    final PersonsModel model = (PersonsModel) getManager();
    ageCS = new CrossSection.Integer(model.getPersons(), Person.class, "age", false);
}
```

The cross-section contains the age value of all the agents in the simulation, to which the Collector has direct access through the *getManager* method. The cross-section is updated through a specific event defined in the schedule by this instruction:

```
public void buildSchedule() {
```

[15] In order to be able to receive and react to the *Stop* event, the model must implement the *EventListener* interface, define the events as *enums*, and specify the method *onEvent* (see section 4.2).

[16] The computation of the cross-sectional statistics can be made with respect to specific sub-groups of the population, after defining appropriate filters. For more information on the use of the statistical package in JAS 2, see the Statistics user guide, available for download at the JAS 2 website.

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

5.4 The *PersonsObserver* class

The Observer graphically displays at runtime the mean and maximum value obtained through the cross-section defined in the Collector.

The *buildObjects* method creates a plotter object that leads to the drawing of historical series of data:

```
public void buildObjects() {
    final PersonsCollector collector = (PersonsCollector) getCollectorManager();
    plotter = new SimulationPlotter("Age", "age");
    plotter.addSeries("avg", new MeanArrayFunction(collector.getAgeCS()));
    plotter.addSeries("max", new MaxArrayFunction.Integer(collector.getAgeCS()));
    GuiUtils.addWindow(plotter);
}
```

The plotter contains two time series obtained by applying the *MeanArrayFunction* and *MaxArrayFunction* respectively to the cross-section defined in the Collector.

The graph update is also managed by events and thus synchronized with the simulation. The *buildSchedule* method defines the frequency of updates:[17]

```
public void buildSchedule() {
    getEngine().getEventList().schedule(new SingleTargetEvent(plotter, CommonEventType.Update), 0,
displayFrequency);
}
```

5.5 Model execution and inspection

To execute the model the *StartPersons* class, which is created automatically by the JAS 2 Eclipse plug-in, is invoked. This class contains the standard *main* method which allows a Java application to run (see section 4.5). In particular, this procedure launches the JAS 2 graphic window, creates a model instance and gives it to the simulation engine. It then creates a Collector and an Observer and recalls the *setup* method of the simulation engine, which has the task of loading the experiment into memory ready to be executed. The window contains a mask for setting the additional model parameters defined in the *Model* class, and a dynamic graph displaying the statistics specified by the Observer (in this example, average and max age in the population).

As already explained (see section 4.5), the *StartPersons* class also permits to launch the model in batch mode (without the Observer) and run multiple model executions with different parameters, in order to explore the space of model solutions.

The *Tools / Database explorer* tab allows to browse past experiments, opening the H2 database management tool and inspecting the data contained therein. By selecting a specific experiment and pressing the *Show database* button the data collected during the simulation can be explored using SQL commands (Figure 6).

[17] For a description of the *Plotter* object, together with other graphical tools, see the specific tutorial available at the JAS 2 website.

Figure 6. Database inspection.

The H2 management tool also allows the table to be exported in CSV format for subsequent import into other statistical tools. By typing in the command:

```
CALL CSVWRITE('C:/test.csv', 'SELECT * FROM PERSON ');
```

the file is exported in CSV format in an H2 window.

6. Concluding remarks

The JAS 2 simulation platform achieves a convergence between agent-based and microsimulation tools. The platform can be assessed both with respect to what it is, and with respect to what it is not. First, JAS 2 is not a tool to speed up simulation execution; rather, its 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 work done in the past by the same authors or by other researchers is crucial. Simulation modeling needs cooperative development. The choice of an entirely open-source tool should also be evaluated in this light.

In the trade-off between efficiency and transparency, we deliberately opted for the latter. However, JAS 2 does not force the user to adopt predefined solutions to the problems faced in simulation modeling. By offering a set of libraries that extend the capability of the standard Java classes, JAS 2 leaves entirely open the possibility of using other libraries and tools, either as an alternative or on top of the JAS 2 toolkit. This is similar to other platforms as MASON and Repast, which are also Java-based and open-source. However, these simulation toolkits leave input/output communication somewhat in the backyard, and are therefore ill-suited for microsimulation modeling.

From a computer science perspective, the main value added of JAS 2 lies in the integration of an object-oriented simulation platform with a relational database, through the use of ORM. Clearly, our approach is an overkill for small-scale agent-based models. Toy models meant to understand relevant mechanisms of social interaction do not generally need relational archives for input and output. However, the use of large-scale agent-based macro models is becoming increasingly popular as an alternative to the standard DSGE approach in macroeconomics. At the same time, our methodological proposal of strictly separating (i) model specification (the agents and their environment), (ii) micro and macro algorithms (the econometric formulas used for predicting outcomes at an individual level and the specific methods used for alignment and matching), (iii) data collection and analysis, could also be useful for dynamic microsimulation modeling. This separation is possible thanks to a strict adherence to an object-oriented approach and a detailed package structure. The prices to pay, with respect for instance to Liam2 or Modgen with their own syntax based respectively on Python and C++, are possibly speed of execution (a

proper testing is still to be done) and a slightly longer learning curve. On the pros side, the possibility to extend the platform in endless directions due to its open-source architecture, the readability that comes with an object-oriented approach especially when the models scale up, and the power and flexibility given by the possibility of storing the underlying data in a relational database. As with most things, diversity is a strength, and in light we hope JAS 2 will be welcomed in the agent-based and dynamic microsimulation communities.

References

De Menten G., Dekkers G., Desmet R., Bryon G., Liègeois P., Wagener R., O'Donoghue C. (2012). "LIAM 2: a new open source development tool for the development of discrete-time dynamic microsimulation models". Paper presented at the Séminaire scientifique international Direction des retraites et de la solidarité de la Caisse des Dépôts, Bordeaux, France, November 15th, 2012.

Gilbert N., Terna P. (2000). "How to build and use agent-based models in social science". *Mind & Society*, 1(1): 57-72

Keller A.M., Agarwal S., Jensen R. (1993). "Enabling The Integration of Object Applications With Relational Databases". *ACM SIGMOD Proceedings*

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*". Kluwer, Dordrecht

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*, Springer, Heidelberg.

Wilensky U. (1999). "NetLogo (and NetLogo User Manual)". Center for Connected Learning and Computer-Based Modeling, Northwestern University.