Agent-based Simulation in AgE Framework

Łukasz Faber, Kamil Piętak, Aleksander Byrski and Marek Kisiel-Dorohinicki

Abstract The chapter introduces AgE framework as a core for constructing agent-based simulation systems. Its features are described against other solutions that may be used in the area of agent-based simulation. The discussion focuses on technical issues—the support for agent-specific services as well as the mechanisms allowing for extensibility and flexibility of the configuration of simulation models and systems. The considerations are illustrated by a simple case study, which aims at showing the differences between AgE and several selected tools for agent-based simulation.

1 Introduction

For certain problems appropriate models may be built and desired research may be performed using available means—without specialized infrastructure (features of uniform motion may be explored using a battery-propelled toy car and a stopwatch, a toy car let down the slope may help in understanding motion with constant acceleration, etc.). However, the research on complex systems (e.g. factory assembly line) and phenomena (e.g. metal casting) often requires great effort that must be put on examining their determinants, defining conditions for experimental runs, and the observed features analysis. Therefore, appropriate strategies of performing experiments must be defined, requiring placing of sensors and reading them, statistical processing of the acquired data and computing their plausibility.

Unfortunately, paraphrasing Heisenberg's uncertainty principle, by introducing sensors into the system (e.g. reading the temperature during casting), the conditions of experiments are changed, and readings of these sensors are affected by their presence. Also 'live' system analysis suffers from constrained possibilities of repeating

Łukasz Faber, Kamil Piętak, Aleksander Byrski, Marek Kisiel-Dorohinicki
AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: `faber@student.agh.edu.pl, {kpietak,olekb,doroh}@agh.edu.pl`

the experiments with exactly the same conditions, which is required for statistical processing of acquired data.

Obviously, there exists a possibility of constructing mathematical models, which allow to formalize certain aspects of phenomena under interest, e.g. the process dynamics based on differential equations or Markov chains [24]. Yet such models are usually quite complex and have limited (though non-disputable) applicability in pure theoretical analyses (cf. [8, 23]). Computer-based models and simulations support the researcher with powerful means to create a virtual environment and conduct experiments with a possibility to repeat the simulations under different conditions, generating vast amounts of data ready to be processed statistically for presentation [28]. Thus, appropriately defined simulation may help in understanding systems or phenomena of interest, prior to performing real-life experiments.

As digitial computers are discrete-time machines *per se*, discrete-time simulation becomes a natural way of defining such models and implementing different supporting tools [3]. Since it was possible to generalize typical elements of such models (e.g. objects, events, actions), dedicated software tools became available, most of them proprietary, but several open-source too [22, 19]. At the same time the idea of agency (e.g. [29]) seems well suited for implementation of heterogeneous complex systems simulations, as the globally controlled algorithm is replaced with local perception and interactions among the agent, its neigbours in the environment and the environment itself. The resulting combination of the discrete and agent-based approach constitutes a base for a wide range of simulation systems (see, e.g., [20, 18, 16]).

In the course of this contribution, an agent-oriented framework *AgE* is presented as a tool supporting the construction of distributed simulation systems. The chapter begins with a short review of available discrete-event simulation frameworks. In the next section, the structure and principles of work of *AgE* are presented. The discussion of different features of *AgE* framework shows that it might be interesting and competitive when compared to other available open-source software. In the end, a case study depicting the application of selected frameworks to simulation of simple inter-agent interaction is presented.

2 Agent-based simulation frameworks

Among a variety of problems (sociological, biological, etc.) requiring simulation approach there are complex processes observed in populations consisting of a huge number of different, possibly autonomous individuals. For such problems macro-scale models may be defined, which allow for understanding the dynamics of the emerged phenomena using appropriate mathematical apparatus, in order to perceive its certain features. Alternatively, mimicking the behaviour of single interacting entities, and observing the emerged phenomena in the whole population may be considered. Such entities, when situated in some environment, capable of perceiving the environment, interacting with themselves and the environment fall under the definition of autonomous agents [29].

Agency brings many improvements into the world of simulation, following the idea of decentralisation of control. Each agent may be autonomous, differently configured, utilising different means of discovering the features of the environment and its neighbours, utilising different algorithms and performing different actions in the system. Because of these, interesting added value may be taken for granted: building complex systems consisting of different, interacting beings will be natural when referring to agents as simulation objects.

As an example, one may consider economic modelling, which can, take into account new and, arguably, more precise characterisations of human beings. This way of modelling economic agents may become an alternative to more traditional mathematical models employed in economics. Those traditional descriptions of human beings normally exclude, for the sake of tractability, fundamental aspects such as qualitative descriptions of the agents' goals and intentions, beliefs and other attributes of human reasoning (e.g., bounded rationality) [26].

To recapitulate: the use of agents opens up new possibilities to introduce models very close to their real equivalents. At the same time the approach of agent-based simulation, because of its inherent logical decomposition and decentralisation of control, allows for building models featuring high flexibility and extensibility. Obviously all these features must be supported by dedicated software tools.

2.1 Technical issues

In agent-based simulation the problem of synchronisation of autonomous entities, usually acting in parallel becomes of vast importance. Classical issues known from the parallel programming [1], such as deadlocks or starvation must be avoided, that becomes a non-trivial task when considering simulations consisting of hundreds or even milions of agents. Technical problems would arise if the agents were implemented as independent threads, especially in large systems, involving distributed processing. Far better seems to inverse the problem of synchronisation, by following so called *phase simulation* approach [21] in which there exists a synchronisation mechanism shared by a group of agents, which takes care of letting each regular agent (taking part in the simulation) do some activities (e.g., perform some query on the system state, change its state, register some action to be later performed by the synchronisation mechanism. In this way, complex parallel programming is changed into a kind of 'round robin' technique that does not pose problems in perceiving parts of the simulation as still autonomous (and acting in parallel, in, *nomen omen*, simulated way). Because of that, agent-based discrete-time simulation frameworks gained the attention of the authors.

Following the requirements of agency, other technical issues influencing the implementation should be considered, to make possible construction (or adaptation) of general-purpose agent-based simulation frameworks fulfilling the user's needs:

- life cycle control (means for definition of an agent being and managing its life cycle in the system),
- communication facilities (means for inter-agent and agent-platform communication making possible interaction between agents and the system, and among the agents, even if they are placed in remote locations, as different computation nodes),
- organisation (possibility of introducing some structures into agent organisation, e.g., groups or even trees of agents), making possible mimicking behaviour of real societes or implementing various divide and conquer-like algorithms),
- distributed computing (for simulations which require running on computer clusters so the total time of experiment is reduced),
- exetensibilty (low coupling on different levels of implementation, following e.g., reusable software components paradigm, that help in modularisation and generalization of the framework application, making possible easy exchange of algorithm, agent, environment parts so that the platform may be easily reconfigured for different experiments, repeating simulations in different conditions, focusing on configuration instead of programming),

One may also consider the platform code status—is the code open-source, up-to-date, cross-platform, making possible to quickly learn provided API, using as a support existing developer's blogs or fora.

2.2 Existing solutions

There exists a plethora of multi-agent frameworks which may be used to support the construction of simulation systems. Some of them are oriented to specific kinds of simulation (see [19, 22]): e.g., simulating of movement of entities with 3D visualisation (see e.g., breve [17]), networking (see e.g, ns2/ns3 [9]), possibility of visual programming (see e.g. SeSam [27]). When looking for universal agent-based simulation frameworks (especially in open-source software domain), one should consider such products as Galatea [12], RePast [20], Mason [18], SystemC [6], SimPy[1]. Other ones are general-purpose agent-based programming frameworks (e.g. JADE [4]) that may be of course adapted to any kind of simulation. MadKit [16] should also be mentioned as a framework for simulating complex populations (following Agent/Group/Role paradigm [13]). An interesting, though not popular framework is Galatea [12] that makes possible utilising HLA [11] infrastructures for running the simulation as federation.

The below-mentioned frameworks were selected as the most promising examples of general-purpose agent-based simulation frameworks in the open-source market.

[1] http://simpy.sourceforge.net/

MASON

MASON is an agent-oriented simulation framework developed at George Mason University. The name refers to the parent institution, as well as derives from the acronym Multi-Agent Simulator Of Neighbourhoods (or Networks) [18]. Mason is advertised as fast, portable, 100% Java based. Multi-layer architecture brings complete independence of the simulation logic from visualisation tools which may be altered anytime. The models are self-contained and may be included in other Java-based programs. Various means for 2D and 3D visualisation, and different means of output are available (PNG snapshots, Quicktime movies, charts and graphs, data streams).

Simulation in MASON consists of a model class `SimState` that composes random number generator and a `Schedule`. An object of `Schedule` class manages many agents, implementing `Steppable` interface, therefore an agent may interact with other ones and the environment by exposing predefined method that will be called by the `Schedule` [18]. The `SimState` may also manage the spatial structure of the simulation with a concept of `Fields` allocating different objects, thus constructing environment in which the agents may be situated.

Programming model of MASON follows basic principles of object-oriented design. An agent is instantiated as an object of a class, added to a scheduler and its `step` method is called during the simulation. There are no predefined communication nor organisation mechanism, these may be realized using simple method calls. There are neither ready-to-use distributed computing facilities nor component-oriented solutions.

First released in 2003, the environment is still maintained as an open-source project, distributed under Academic Free license (ver. 3.0). The current version (16.0) was released in the end of 2011.

RePast

RePast—Recursive Porous Agent Simulation Toolkit—is widely used agent-based modeling and simulation tool. Repast has multiple implementations in several languages and built-in adaptive features such as genetic algorithms and regression [20]. The framework utilizes fully concurrent discrete event scheduling, HPC version also exists [10]. In Repast 3, there are many programming languages interfaces (e.g., Java, Logo dialect, .NET languages, Lisp dialect, Prolog, Python). Logging and graphing tools are built-in. Dynamic access to the models in the runtime (introspection) is possible using graphical user interface. There are predefined libraries for different methods of modelling and analysis available, e.g., neural networks, genetic algorithms, social-network modelling, GIS support.

The implementation of a simulation system in RePast 3 is realized in a similar way as described for MASON. The class suitable for simulation should extend `SimpleModel` class, and contain appropriate implementation of `step()` function that will be called by the scheduler. It is to note that proper construction of the class' attributes allows to edit them using introspection mechanism supported by RePast GUI. The simulation may be even interrupted, available parameters may be changed and the simulation may be carried on.

Repast 3 consists of different implementation of the platform (Repast J—Java-based, Repast.NET—MS .NET and Repast Py—Python). It has been renowned for a long time, however, recently Repast 3 has been superseded by its next stage development called Repast Simphony (Repast S) bringing newly developed GUI, with some significant changes into the programming paradigm.

In Repast Simphony, graphical programming mode has been introduced, therefore a model may be constructed according to Rapid Application Development approach to software construction and the generated code may be easily integrated with Java and Groovy components. The execution environment supports exporting the simulation results to many popular external tools such as R, Weka or MATLAB. Sophisticated 2D and 3D visualisation features make possible integration with JUNG and GIS. The Repast Simphony inlcudes various libraries supporting genetic algorithms, neural networks, regression, random number generation and other mathematical tools.

In Repast Simphony, there has also been a new organisation concept called 'context' introduced. It generally consists of a group of unorganized agents (they may be organized using so-called projections) and may create a hierarchical structure (context can have many sub-contexts and so on). This idea affects the perception of agents in such way, that an agent in the sub-context also exists in the parent context, but the reverse is usually not true.

The latest (Simphony 2.0 beta) version of this open-source project, licensed according to 'new BSD' license, has been released in the late 2010.

MadKit

MadKit is a modular and scalable multi-agent platform written in Java aimed at modelling different agent organisations, groups and roles in artificial societies. Extensive GUI is available, as well as different programming languages for agents definitions (e.g., Python, Scheme, Jess, BeanShell) [16].

MadKit is built based on so called Agent/Group/Role organisational model [13] utilising plugin-based architecture. The architecture of MadKit is based on microkernels which provide only the basic facilities: messaging to local agents, management of groups and roles, launching and killing of agents. Other features (remote messages, visualisation, monitoring and control of agents) are performed by agents. Both thread based agents and scheduled agents for multi-agent based simulation may be developed. Different execution contexts are available: JSP, Applet, Console mode, GUI desktop, etc.

Threaded agents participating in simulation inherit from the `Agent` class and need to define at least the `live()` method that specifies their behaviour. This method represents the main loop of the thread. Non-threaded agents, inheriting from the `AbstractAgent` class, may also be used but they require a scheduling agent that can control their life cycle (e.g., which methods should be called at what time). Besides schedulers there is also one more kind of special agents, called *watchers*, inheriting from the `Watcher` class. They can use a set of *probes* to inspect parameters of other agents or the whole communities. All agents share also common life cycle calls: `activate()` run during agent initialisation and `end()` that is called at the end of the life-cycle.

Simulation does not require any particular structure or model to run. However, as mentioned earlier, it is possible to add an arbitrary scheduler or create complex agent communities and relations. Agents can locate other agents that handle a specific role (or many roles) or are members of a particular group. They can communicate with each other using these roles or group membership (i.e. using broadcast messages) and react to them.

MadKit provides an environment (in the desktop-like form) that allows for rapid creation and manipulation of agents with tools like a code editor, an agent designer or an agent observer that can, for example, log all exchanges of messages. There is also some support for binding GUI elements (graphical representations) to agents. User can replace parts of the environment and extend it with system agents that can use hooks in the micro-kernel to receive notifications about system events (publish-subscribe model).

The latest stable (4.2) version of this open-source project, licensed according to GPL license, has been released in 2008. Currently, alpha releases of version 5 are also available.

3 Agent-based simulation in AgE

AgE environment[2] is being developed as an open-source project at the Intelligent Information Systems Group of AGH-UST. *AgE* provides a platform for the development and execution of distributed agent-based applications—mainly simulation and computational systems.

Fig. 1 presents an overview of a system based on *AgE* platform. A user constructs the system by providing an input configuration in XML format. The configuration specifies the simulation structure and problem-dependent parameters. After the system start-up, the environment (agents and required resources) are instantiated, configured and distributed amongst available nodes where they start performing their tasks. Additional services such as name, monitoring, communication and topology service are also run on these nodes to manage and monitor the computation. The output of the simulation is problem-dependent and may be visualized at run-time by dedicated graphical tools and interpreted by the user.

[2] http://age.iisg.agh.edu.pl

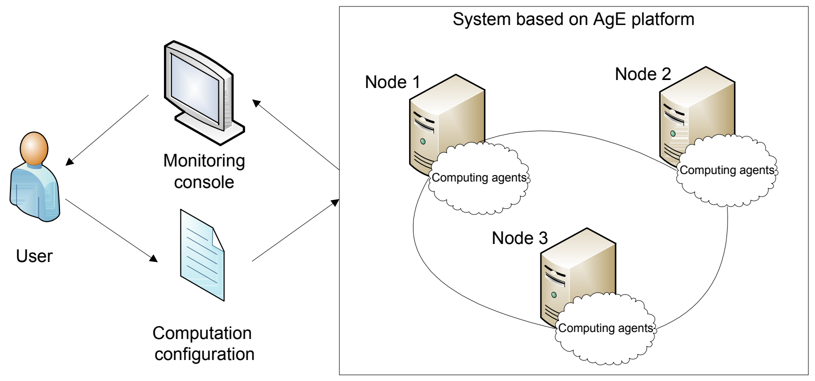

Fig. 1 AgE system overview

3.1 *Structure and execution of agents*

A simulation is decomposed into agents, which represent individuals or parts of or whole populations. Agents are structured into a tree with virtual root agent (as shown in Fig. 2) according to the simulation decomposition. The top level agents (called *workplaces*) along with all their children can be distributed amongst many nodes.

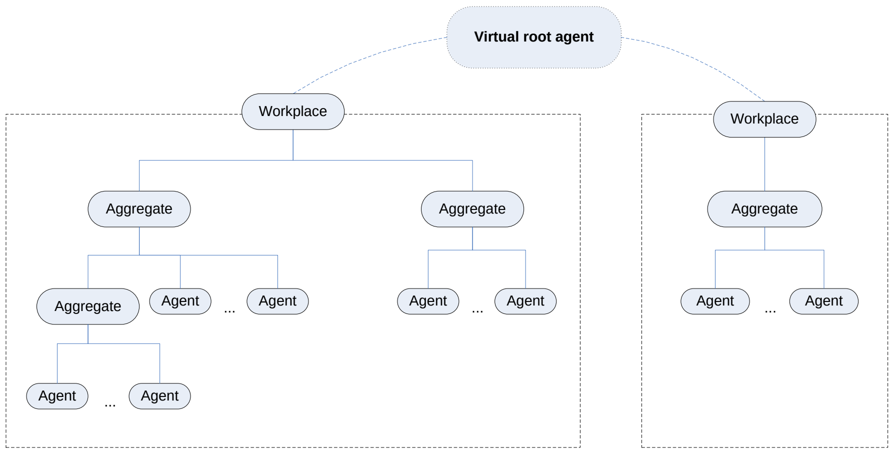

Fig. 2 Agent tree structure

Agents can have named properties, which are features of an object, which can be referenced during run-time by its name in order to access, modify or even monitor its value. Properties are defined by annotating fields or methods of agents classes with dedicated Java annotations [7]. Each agent exists in an environment, defined by the parent agent, which provides a context of agent's processing. With the use of the environment, agents can communicate with their neighbour agents via messages, acquire specific information about them via queries, or even request them to perform specific actions.

It is assumed that all agents at the same level of the structure are being executed in parallel. The platform introduces two types of agents: thread-based and simple. The former are realized as separate threads so that the parallel processing is managed by Java Virtual Machine (similarly to JADE platform). Such agents can communicate and interact with neighbours via asynchronous messages. However, a large number of such agents would significantly decrease the performance of a simulation because of frequent context switching and raises synchronisation problems as discussed in Section 2. Therefore, following the concept of *phase simulation*, the notion of simple agents is introduced. The execution of simple agents is based on *steppable* processing which is to simulate pseudo-parallel execution of agents' tasks. Two phases are distinguished:

- Execution of tasks related to computation semantics in the `step()` method. In case of an aggregate agent all it's children perform their steps sequentially. While doing so, they can register various events, which may indicate actions to perform or communication messages, in the parent aggregate.
- Processing of events registered in an event queue. Since events may be registered only in agents that possess children, this phase concerns only aggregate agents.

The described idea of agents processing ensures that during execution of computational tasks of agents co-existing at the same level in the structure (agents with the same parent), the hierarchy remains unmodified, thus the tasks may be carried out in any order. From these agents perspective, they are processed in parallel. All changes to the agent structure are made by aggregates during processing of the events that indicate actions such as addition of a new agent, migration of an agent, killing an already existing agent, etc. They are visible for agents while performing the next step.

3.2 Actions

The environment of simple agents determines the types of actions which may be ordered by child agents. It also provides concrete implementations of these actions and thereby supplies and influences agents' execution. Thus actions realize the agent principle of goal level communication [5], because agent only lets the environment know what it expects to be done but it does not know how it will be done.

Simple agents request their parent aggregates to execute actions during an execution of a step of processing. Then, all of actions are executed sequentially (in order of their registration) by the aggregate after all children agents finished their operations.

Because some of the actions can significantly change the environment (for example removal or migration of an agent) so that the other actions would become invalid, the following phases have been introduced:

1. initialisation (`init`), when target addresses are verified,
2. execution (`main`), when the real action is executed,
3. finalisation (`finish`), for performing activities that could not be executed during the main phase (e.g. removal of an agent when other agents could refer to it).

All changes of agents structure that can influence execution of other registered actions are performed in the finalization phase. As a result, performing an action in the execution phase is safe. In the initialization phase actions can perform some preparetion activities that are required by other actions.

Two types of actions exist:

- Simple actions that can define only one task to be performed on only one agent.
- Complex actions — they are containers for other actions and can hold a tree-like structure. Actions wrapped by them are executed in a well-defined order and allows to create more complicated scenarios like an exchange of resources, when the separate component actions are required for getting a resource from one agent and for delivering it to another.

Most simple aggregate actions are defined as methods in a class of an aggregate agent and the default aggregate implementation provides some actions out-of-the-box:

- adding of a new agent,
- moving an agent to another aggregate,
- death of an agent,
- cloning of an agent.

Moreover, users can extend the platform with any actions they need. These actions can be created as strategies bound to the aggregate using the configuration of the platform. They allow to extend functionality of the platform in an easy way but have a downside of not having the possibility to refer to private members of the aggregate. Decision of how to execute such actions is made by the parent agent who resolves proper action implementation according to *Service Locator* design pattern [2].

3.3 Life-cycle management

The lifecycle of an agent consists of the following phases:

1. Construction — when a constructor of agent class is called.

2. Initialisation of the object dependencies and properties — when the `init()` method is called; at this point the agent has all its dependencies injected by the component framework based on dependency injection pattern mechanism. Also its properties are initialized using the component framework or by agent itself. For example at this stage, an agent generates an address
3. Initialisation of the environment — the moment when the parent of the agent calls the `setAgentEnvironment()` method. At this point the agent can use mechanisms that requires the existence of the local environment i.e. actions, queries, messaging.
4. Finalisation of the agent — the `finish()` method. The agent should finish its operation at this point.

Threaded agents additionally provide the `run()` method, called by the Java Virtual Machine after their dedicated thread was started. At this moment they can start the main loop of their execution.

The full lifecycle of the simple agents is shown in Fig. 3. Simple agents need to provide an implementation of the step. It is done in the `step()` method. This operation is called in an arbitrary order by the parent aggregate on every agent it contains. The actual execution from the point of view of the whole tree of agents is performed in the postorder way: firstly the aggregate lets children to carry out their tasks and only after they finished them it executes its own tasks.

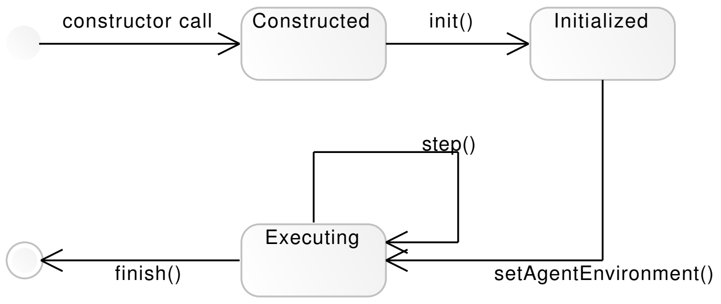

Fig. 3 A life-cycle of the simple agent shown as a state diagram

During the execution of the step, the simple agent usually needs to perform following actions:

- receive and send messages,
- execute queries,
- execute a part of the computation,
- order actions for the parent.

After iterating over all children, the aggregate needs to process the event queue. These events are usually actions requested by the children.

3.4 Communication facilities

The platform allows for all agents to have a unique addresses, which allow for their identification and supports inter-agent communication. The particular property of being globally unique is guaranteed by a structure of the address. As shown in Fig. 4, the agent address comprises of three components: an UUID[3], a node address, and a name. Two former parts identify an agent in the platform instance and the last one is a replacement for an UUID provided for the user convenience (for usage in a user interface or logs).

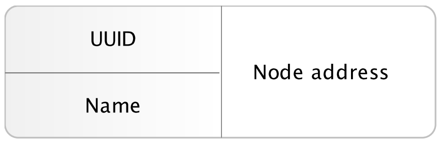

Fig. 4 Components of an agent address

An address is usually obtained by an agent during the initialisation of the component dependencies (see page 10 for the explanation of the agent life-cycle). It is done by requesting a new address object from the `AgentAddressProvider` instance that is a local component of a node.

Communication via message passing

Agents located within a single aggregate can communicate with each other via simple messages.

Interfaces used in messages are shown in Fig. 5. A message defined by the `IMessage` interface consists of a header and payload. The header, as defined by the `IHeader` interface must specify a sender of the message (usually the agent that created the message) and its receivers. The payload is simply a data of any (serialisable) type that is to be transported.

Receivers are defined using selectors. They offer a possibility to define receivers with the *unicast*, *broadcast*, *multicast* or *anycast* semantics.

In the case of simple agents, sending and delivery of messages is performed by an aggregate agent. The sender adds a message event to its parent queue. The parent handles it by locating all receivers and calling a message handler on each of them. These messages are placed on a queue and can be received by the agent during its next step.

Thread-based agents use a similar queue of messages but are not restricted by the execution semantics and can inspect it at any point of time.

[3] Universally Unique Identifier

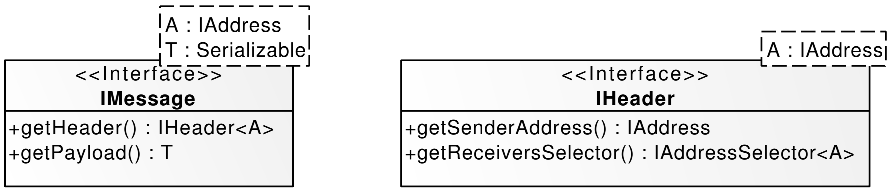
A : IAddress
T : Serializable
<<Interface>>
IMessage
+getHeader() : IHeader<A>
+getPayload() : T
A : IAddress
<<Interface>>
IHeader
+getSenderAddress() : IAddress
+getReceiversSelector() : IAddressSelector<A>

Fig. 5 Overview of interfaces used in messaging

Query mechanism

Queries offer a possibility to gather, analyze and process data located both in local and remote (from the point of view of the query executor) components.

The diagram in the Fig. 6 shows base classes and interfaces of the query mechanism along with their interconnections. The central point of this mechanism is the `IQuery` interface. It provides only one method: `execute()`. A query, as defined by this interface, is an action performed on a target object that leads to creation of query results. Specific implementations define a relationship between the target and results.

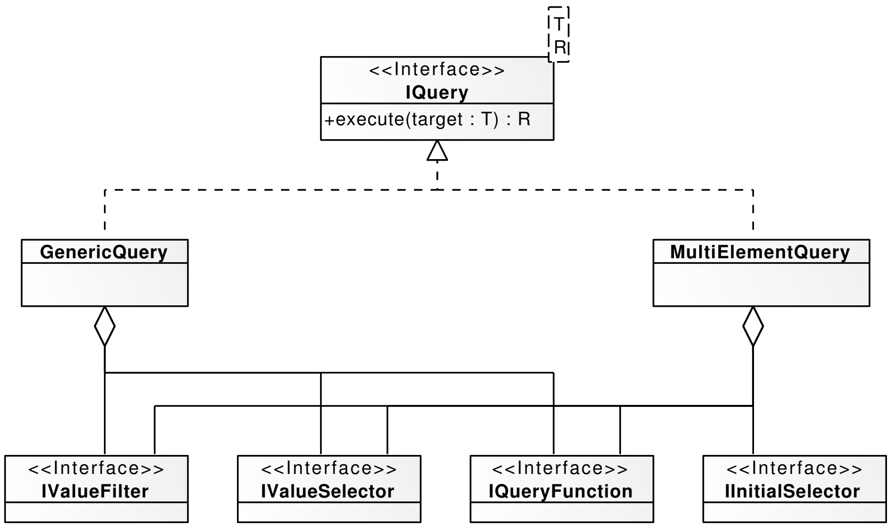

Fig. 6 Overview of queries base classes and interfaces

On the top of this interface and definition, a simple, declarative, yet extensible query language is built. Queries are implemented as (`GenericQuery` and `MultiElementQuery` classes in the diagram 6. It allows the user to perform tasks like: computation of the average value of some chosen properties from the agents in the environment, select and inspect arbitrary objects in collections and much more.

The following operations are defined:

- Preselection of values from the collection. It is only available if the query is performed over an iterable type instance. Its task is to select some (e.g. first ten or random) of objects without usage of the object-related information.
- Filtering by a value. This is an operation similar to WHERE clause in SQL.
- Selection of values. It can select specific fields from objects and it shows some similarities to the SELECT operation from SQL. If this operation is not defined then whole objects are selected.
- Functions working on an entire result set. They can remove, add or modify elements.

Operators are defined as implementation of specific interfaces (one for every operation, as shown in Fig. 6). They are presented to the user as static methods (e.g. `lessThan()`, `pattern()` etc.).

A query is built by specifying following properties:

- A type of the target object (the object passed as an argument to the `execute` method).
- A type of results.
- In the case of collections — a type of elements in a collection.

Such an exhaustive specification is required because queries rely on these pieces of information to control correctness of operators used by the user (with the usage of Java generics). Moreover, queries in AgE are built without the knowledge of the target object (it is in opposition to many similar mechanisms like LINQ[4]).

After that, an operation of the query is specified using aforementioned operations. The execution of the query is carried out by calling the `execute()` method.

The following Java code shows a simple example of how a query can be created and executed. In this case a collection of strings is queried.

```
CollectionQuery<String, String> q =
    new CollectionQuery<String, String>(String.class);
q.from(first(10))
    .matching(anyOf(
        pattern("li.*"),
        pattern("lorem[Ii]psum")));
Collection<String> results = q.execute(someList);
```

It can be noticed that queries definition uses the *fluent interface* pattern with specific operations being composed from static methods.

This approach of declaring a query without the knowledge of the target is additionally useful because it allows to execute a single query many times (possibly with caching the results or operations) or to delegate queries to be executed in another

[4] http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

location. The query delegation is actually often used within the platform during the operation of querying an environment of a parent of an agent. This mechanism is essential for performing the migration of agents.

The other side of the queries mechanism is the extensibility offered to the user on many levels. It is possible to create completely specialized queries (by implementing the `IQuery` interface), extending the described declarative mechanism or even define in-line operators when creating a query. This elasticity of queries was very important because of performance requirements resulting from some applications of the platform. An approach was adopted, that the user is able to provide much faster solutions for his specific problems.

In some cases it is also useful to let know a queried object about a query being executed on it. For this reason the interface named `IQueryAware` was created. By implementing it any object can communicate to the query that it wants to be notified about some events related to the execution. Currently, two events are supported: initialisation and finalisation of the query.

The last part of the queries mechanism is caching. The platform offers a possibility to build a cache of query results. Its expiration time is based on the workplace step counter. This cache works as a wrapper to the query (and as such is an implementation of the `IQuery` interface). During the execution it checks whether stored results expired and possibly executes a real query replacing old results.

3.5 AgE component framework

The platform provides dedicated component framework, which is built on the top of an IoC container. It utilizes PicoContainer framework[5]—a popular open-source Java implementation of IoC container that can be easily extended and customized.

Both agents and strategies are provided to the platform as components. Their implementation classes can define named dependencies to other components (i.e. other agents, services or any other dependent classes) and simple properties that hold for example problem-dependent values. The dependencies definition for a component type, together with class's public methods (treated as component's operations) may be perceived as requirements closely related to component contracts (as proposed by Szyperski [25]).

The process of assembling a system is divided into two main phases. In the first one, the input configuration is read from XML file with well-defined structure[6] and further transformed into object configuration model, structure of which is shown in Fig. 7. A `ComponentDefinition` instance describes a single component and contains data such as it's name, type (which is the name of a class) and scope, which is to determine if a new component will be created for each request (*protoype* scope) or only once during the first request (*singleton* scope). The definition also

[5] http://www.picocontainer.org

[6] http://agh.iisg.agh.edu.pl/age-2.3.xsd

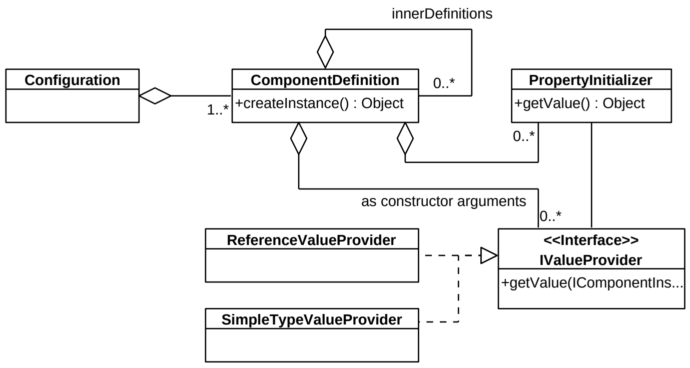

Fig. 7 Object configuration model

specifies the constructor arguments, which are implemented as `IValueProvider` objects and used during constructor-based injection [14], as well as property initializers, responsible for initialising component properties with reference or simple values. Moreover, the definition contains `createInstance` method which creates a new instance of a described component with initialized dependencies (this process is described below). Component definitions may form hierarchical structures (via `innerDefinitions`). If a definition is a child of another one, it is said to "exist in the context of the outer definition" and is visible only for it's parent and other parent's children (siblings). Validation of the model, performed during processing of the input configuration, allows for detecting errors such as unresolved dependencies, non-existent components or incorrect property definitions.

In the next phase of system assembly process, a hierarchy of IoC containers is built according to a structure of component definitions. For each definition a dedicated adapter (`CoreComponentAdapter`) is created and registered in a container as shown in Fig. 8. Moreover, the adapter implements the interface, which defines methods for retrieving instances of components by name or type — `IComponentInstanceProvider`.

When a request for a component instance is directed to the container, it locates appropriate component adapter (using given name or type) and delegates the request further, to it. The adapter calls the associated component definition's `createInstance` method, which is responsible for creating a component instance. While instantiating a component the component adapter retrieves instances of dependent components from associated IoC container (or its parent), and the loop whole process starts again. In the case of simple types, a value is kept directly in a value provider object and is returned on a request. The whole process is repeated until all dependencies are resolved and then the fully-initialized component instance is returned to the client.

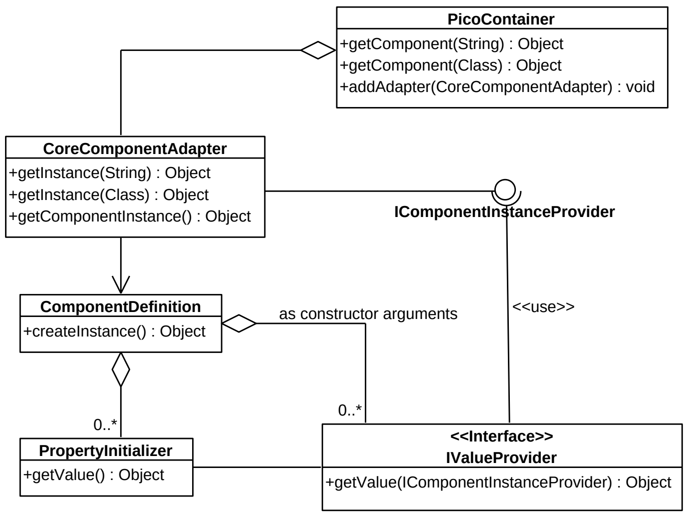
PicoContainer
+getComponent(String) : Object
+getComponent(Class) : Object
+addAdapter(CoreComponentAdapter) : void
CoreComponentAdapter
+getInstance(String) : Object
+getInstance(Class) : Object
+getComponentInstance() : Object
IComponentInstanceProvider
ComponentDefinition
+createInstance() : Object
as constructor arguments
<<use>>
0..*
0..*
PropertyInitializer
+getValue() : Object
<<Interface>>
IValueProvider
+getValue(IComponentInstanceProvider) : Object

Fig. 8 Dependency Injection in AgE platform

The presented mechanism gives a possibility to build various structures of agents with their dependencies and initial properties values based on the input configuration.

3.6 Node architecture

The simulation is executed in a distributed environment comprised of nodes connected via communication services. Each node is a separated process being executed on a physical machine. Nodes are responsible for setting-up and managing an execution environment for agents performing a simulation, as well as assuring communication and collaboration in distributed environment.

The main part of the node is a service bus that realize *Service Locator* design pattern. The bus is realized by *AgE component framework*, which utilizes IoC container to create and initialize an object that is a run-time instance of a service. Services are being registered in the container by the node boot-strapper or other services, based on component definitions, created using API or read from XML configuration file. A reference to a service instance can be acquired by service name or type via `IComponentInstanceProvider` interface.

The node distinguishes stateless and stateful services. The former offer functionality dependent only on parameters given in method call, therefore they does not hold any state and are always thread-safe. They are created on demand (at the first reference) and than their instances are cached in the container.

On the other hand, an instance of stateful service can hold data that influences its behavior. Such services implement `IStatefulComponent` interface, which introduces `init` and `finish` methods, called by the service bus while creating and destroying a service instance. Instantiation and initialisation is performed during node start-up. Stateful services can be also realized as threads, that are started in `init` method and finished asynchronously while destroying the service.

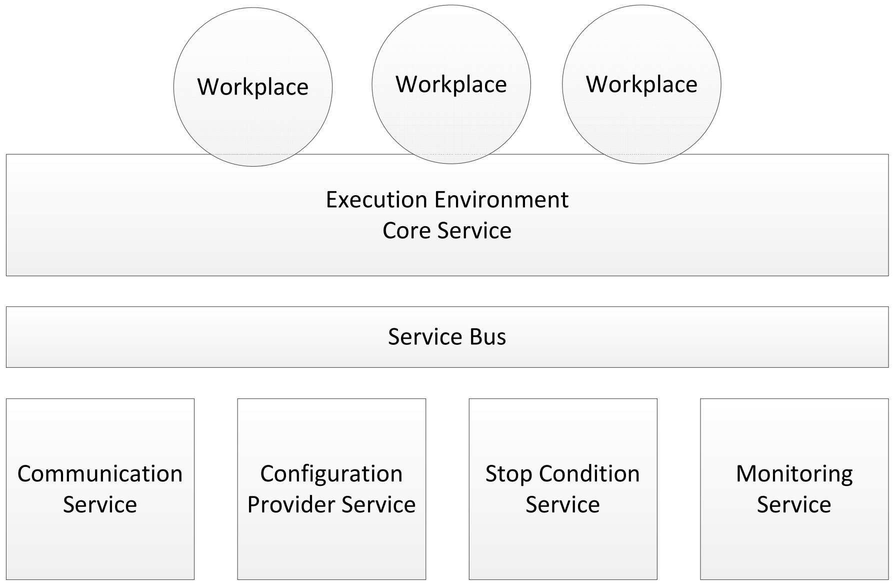

Fig. 9 Example node with services

Fig. 9 shows an example node with registered services. The figure distinguish the main service (called *core service*), which constitutes an execution environment for agents, that provides functionalities such as global addressing schema, communication via message passing, query mechanism, life-cycle management.

This service also plays role of a proxy between agents and other services. Various services provide functionalities related to infrastructure (e.g. communication and configuration provider services), simulation (e.g. stop condition service) or external tools (e.g. monitoring service, which collects and stores simulation data for a visualisation application).

In one distributed environment particular nodes can have different responsibilities such as an end-user console, monitoring, management, and at last execution nodes. The role of a node is specified by the configuration of services plugged into its bus. Also, one can imagine a platform comprised only from a single node that works without any communication services[7].

[7] Such configuration is often used for test purposes.

3.7 Virtual execution environment

The platform introduces *a virtual execution environment* in distributed systems that allows for performing operations involving top level agents (workplaces) located on different nodes without their awareness of physical distribution. Such operations are executed by the core service according to *Proxy* design pattern [15]. The service uses the communication service to communicate with core services located on other nodes. This constitutes *a global name space* of top level agents in the distributed environment. In other words, the virtual execution environment can be perceived as a realisation of *virtual root agent*.

The name space of agents can be narrowed by introducing *agent neighbourhood* that defines visibility of top level agents. An agent can perform operations only on agents from its neighbourhood. The neighbourhood is realized and managed by a topology service (Fig. 10). This allows for creating virtual topologies among agents on the top of the distributed environment. Various topology strategies such as ring, grid, multi-dimensional grid can be applied in simulations.

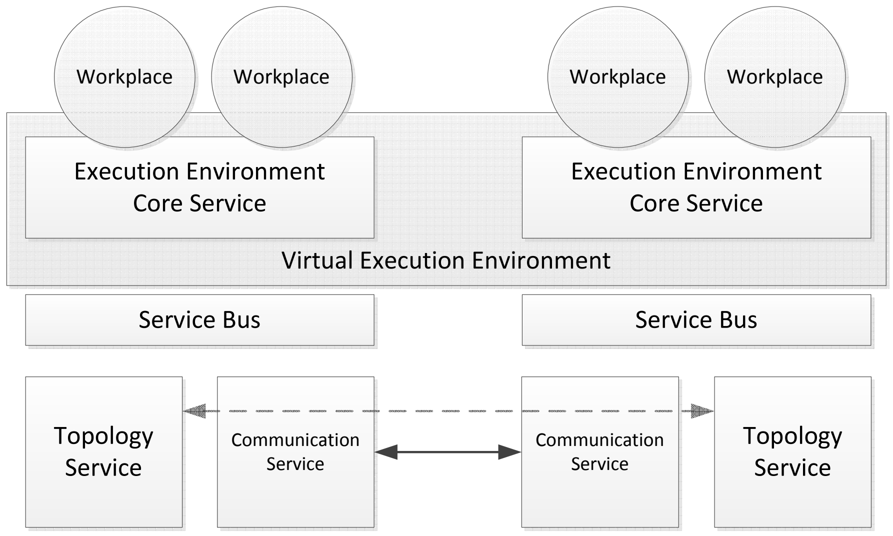

Fig. 10 Virtual execution environment

4 Implementation of sample case study in selected platforms

To compare most basic mechanisms provided by chosen platforms a scenario needs to be defined, to be reviewed in every system. As ‘most basic’ we understand following elements (they were described in the section 2.1):

- life cycle of agents,
- organisation of agents,
- communication between agents (with possibly distributed data or interactions),
- level of coupling in the case of replacing an agent behaviour.

The (very simple) scenario that uses all of this elements can be described as follows:

1. Create and initialize a simulation with two agents.
2. Start the simulation. The agents should locate each other and start exchanging messages saying ‘hello’.
3. Stop the simulation.
4. Change the implementation (behaviour) of the second agent (however, it still should send messages to the other agent).
5. Start the simulation again.

The diagrams 11 and 12 show interactions between these two agents before and after replacing the second one implementation.

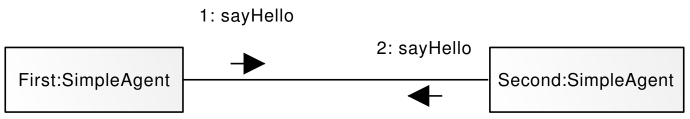

Fig. 11 First phase of the scenario.

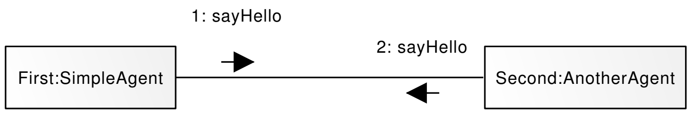

Fig. 12 Second phase of the scenario.

4.1 MASON

We have tested the scenario against the version 16 of the MASON.

Life cycle of agents

Simulation in MASON is created by inheriting from the `SimState` class. The execution of the simulation is started by calling the `doLoop()` method.

Agents are created either by implementing one of child interfaces of the `Steppable` interface, that requires a definition of the `step()` method. An instantiated agent needs to be scheduled to execute. It is done by calling one of methods of the `Schedule` instance located inside the simulation. For example, the following code schedules an agent to run repeatedly every unit of time:

```
Steppable agent = ...;
schedule.scheduleRepeating(agent);
```

Organisation

Agents may create a social network within a simulation. A social network, in MASON terms, is an undirected graph (or multi-graph) whose edges may be labeled and weighted. It opens possibilities to create arbitrarily complex relations between agents.

Moreover, MASON provides models for spatial organisation of agents. Using the `Continuous2D` class a plane for agents to be placed in could be created. Then, we could obtain agents to interact with on the base of their distance from the requesting agent:

```
Bag neighbours =
    space2d.getObjectsWithinDistance(myPosition,
    distance);
```

Where `myPosition` is a position of the agent and a distance is a number of the `double` type.

Communication

MASON does not provide any specific communication capabilities to agents. There is no distribution support either. Thus the developer needs to implement its own solution.

In the simplest case, communication can be realized with the usage of methods calls. It is a reasonable solution due to the fact, that the agent usually obtains a reference to its potential communication partners in some way (e.g. the `getObjectsWithinDistance` returns a bag of real agent objects). For example, saying ‘hello’ to all neighbours could be implemented as:

```
Bag neighbours =
    space2d.getObjectsWithinDistance(myPosition,
    distance);
for(int i = 0; i < neighbours.numObjs; i++)
{ if(neighbours.objs[i] != this) {
     ((Agent)neighbours.objs[i]).sayHello(this);}
}
```

A reference in the `sayHello` method can be used by the partner to respond to the call.

Distribution could be also added to the system but it would require the user to develop a complete solution (probably specific to the simulation) that would allow more-or-less transparent communication between separate MASON instances.

Coupling

A change of a behaviour of an agent is costly. MASON does not provide any built-in features to easily replace implementations of simulation objects (e.g. there is no configuration mechanism that would allow to define a specific implementation for a particular simulation run). Code (of an agent or the simulation initialisation) need to be changed and recompiled in order to use a new behaviour.

On the other hand, MASON provides utilities to build a GUI. One can make use of them to provide a way for the user to load new classes or change simulation parameters. However, it still needs to be programmed by the developer of the simulation.

4.2 RePast

We base this description on RePast Simphony 2.0 beta as this is a version currently recommended to use by developers.

Life cycle of agents

An agent in RePast is created as POJO — there is no requirements directed towards its implementation. To be able to run in the simulation it needs to have one of its method annotated with a scheduling plan. It is possible to use, for example, an active scheduling in the form of the `ScheduledMethod` annotation or the reactive one — the `Watch` annotation.

We would like to have an agent to execute in every step of the simulation since its beginning:

```
class Agent {
  ...
  @ScheduledMethod(start=1, interval=1)
  public void performStep () {
    ...
  }
}
```

The simulation is initialized and built by a `ContextBuilder` implementation that should be provided by the developer. It's implementation is dependent on the specific use-case but RePast provides a default version that behaves like a collection (i.e. `java.util.Collection`) of agents.

Organisation

Similarly to MASON, RePast allows for spatial organisation of agents on multidimensional continuous spaces or grids (however, there is no simple way to obtain neighbours of the object). Moreover, there is a graph organisation model that allows to construct arbitrarily complex networks of agents. It can be created in the context using:

```
NetworkBuilder<Object> builder = new
    NetworkBuilder<Object>("agents", context, true);
builder.buildNetwork();
```

And then, the agent, can obtain associated agents using (the graph is directed):

```
Network<Object> net =
    (Network<Object>)context.getProjection("agents");
for(Agent agent : net.getSuccessors(this)) {
  ...
}
```

Several other projections, like GIS[8], are also available.

Communication

RePast does not offer any built-in messaging or interaction mechanisms beside plain method calls. As in MASON, it is possible to build a specific communication protocol for the particular simulation.

[8] Geographic information system

On the other hand, RePast has a support for the computation distribution using GridGain[9] or Terracota[10] plugins.

Coupling

RePast is similar to MASON also in this aspect. Agents depend on each other's interfaces and a simulation is constructed in a compiled builder. However, there is a support for specifying model parameters in configuration files (`parameters.xml`). Some components of a scenario may also be specified (and thus easily replaced) in the configuration.

4.3 MadKit

We used the version 4.2.0 of the project as it seems to be recommended release (although there are alpha releases of the version 5).

Life cycle of agents

Usually agents in MadKit are implemented by inheriting from the `Agent` class, representing a threaded agent. The main behaviour of such an agent can be provided by its creator in the `live()` method. Additionally, actions to be executed on agent initialisation and finalization can be specified by implementing `activate()` and `end()` methods.

In the discussed scenario a messaging in a do-while loop guarded by some condition could be simply implemented:

```
public class SimpleAgent extends Agent {
  ...

  public void live() {
    do
    {
      exitImmediatlyOnKill();
      ...
      pause(100);
    } while(someCondition);
  }
}
```

[9] http://www.gridgain.com/

[10] http://www.terracotta.org/

The `exitImmediatlyOnKill()` call is a way for an agent to check that it should stop its operation.

Furthermore, MadKit also allows to create agents with more general scheduling semantics thanks to `AbstractAgent` and `Scheduler` classes. The former is used as a base class for non-threaded agents and the latter is a scheduling agent that can execute methods of other agents according to the user specification, with the usage of so-called activators.

Organisation

MadKit offers a possibility for agents to organize into groups. An agent may want to become a member of some group and have a specific role within it. It can be done by firstly locating a specific group and then requesting a role with the `requestRole()` method or alternatively, by creating a completely new group. It can be done from the mentioned earlier `activate` method, called when the agent is started:

```
public void activate() {
  ...
  if (!isGroup("simple-exchange")) {
    createGroup(true, "simple-exchange", null, null);
  }
  requestRole("simple-exchange", "first-agent", null);
}
```

Firstly, the condition is checked, whether the group interesting for us exists and, if not, the agent creates it. Then, it tries to join this group with the role *first-agent*.

Communication

Mechanisms for the local messaging is provided directly in the base class of an agent as few versions of the `sendMessage()` method. In the discussed case we want to exchange messages between two agents. One can choose between two approaches: explicitly locating a communication partner and obtaining its address or by using its role name. The latter has an advantage of decoupling agents interactions from concrete agents instances — i.e. the partner can be easily replaced without introducing any change in the other agent.

```
Message message = new SpecialMessage();
sendMessage("simple-exchange", "second-agent",
    message);
```

Received messages are stored in a queue and can be obtained when needed by a call to the `nextMessage()` method.

It is also possible to use a distributed communication. However, this feature is not directly built into the system. MadKit follows an approach of the ‘agentification’ of services. The user has to introduce an agent that will be able to handle routing and forwarding of non-local messages. Such an agent needs to handle the *communicator* role in the *system* group. When a message that is non-deliverable within a local environment is sent, this agent will receive it and should forward it to another remote communicator.

Coupling

Coupling between agents is low, as most of the interactions is performed using a built-in messaging system that reduces a need to know each other agent interfaces. Most of the services are also built as agents (due to aforementioned 'agentification') and they are often identified by a specific role in the agent community and not by an interface. Thus, it is possible to change parts of the system as easily as casual agents.

MadKit GUI allows for simple changes to behaviour using a built-in code editor, so it is possible to stop the simulation, modify the agent code and rerun it. Similarly, agents can be added to or deleted from the simulation during runtime.

4.4 AgE

We have based this analysis on the version 2.5.0 (current stable) of the platform.

Life cycle of agents

As described on the page 10 the life cycle of an agent is well-defined. However, in most cases only the `step` method must be provided by an agent creator. This method is called once in every step of the computation.

AgE requires an agent to implement the `IAgent` interface. It also provides a default implementation of it in the `SimpleAgent` class that leaves only the aforementioned `step()` method unimplemented.

Organisation

Agents are organized in a tree structure. In this scenario three agents should be used: a workplace (that is a root of a tree) and two leaf agents on the second level of the tree. This structure is defined in the configuration file:

```
<agent name="workplace"
   class="org.jage.workplace.IsolatedSimpleWorkplace">
  ...
  <list name="agents">
```

```
    <agent name="agent1"
       class="org.jage.example.SimpleAgent">
      ...
    </agent>
    <reference target="agent1" />

    <agent name="agent2"
       class="org.jage.example.SimpleAgent">
      ...
    </agent>
    <reference target="agent2" />
  </list>
  <property name="agents">
    <reference target="agents" />
  </property>
</agent>
```

We define the workplace (which is represented by the built-in `IsolatedSimpleWorkplace` class) and, within it, a list of two agents of the type `SimpleAgent`.

Communication

AgE provides a specialized mechanism for sending messages. However, if one wants to send a message to a specific agent, he needs to obtain its address first (alternatively, an agent can send a broadcast message). It is usually done by querying the environment of the agent.

A sample query may look like this:

```
AgentEnvironmentQuery<SimpleAgent, IAgentAddress>
   query = new AgentEnvironmentQuery<SimpleAgent,
   IAgentAddress>(IAgentAddress.class);
query.select(agentAddress());
Collection<IAgentAddress> answer =
   queryEnvironment(query);
```

The part responsible for obtaining addresses of agents is the `agentAddress()` selector. The call to the `queryEnvironment()` method executes a query over the calling agent's environment (in which its partner is located).

Then, the agent can send a message:

```
Header<IAgentAddress> header = new
   Header<IAgentAddress>(getAddress(), new
   UnicastSelector<IAgentAddress>(receiverAddress));
Message<IAgentAddress, String> textMessage = new
   Message<IAgentAddress, String>(header, "hello");
sendMessage(textMessage);
```

At the moment AgE does not provide any means to distribute computation. However, work on this topic has already been started.

Coupling

In AgE, by applying separation of interfaces and providing communication and queries mechanism, agents, their dependencies and platform elements are very low coupled. Moreover, utilizing components techniques gives the wide opportunities to easily create flexible and customizable simulation systems. Particulary, agents structure, their behaviour or even simulation parameters can be easily changed by providing modified runtime configuration. Such changes do not require any code updates so that the platform components and libraries do not have to be recompiled. One can also imagine that multiple similar simulations can be run in batch-mode (one after another) without any user inference.

4.5 Summary

We have reviewed four different multi-agent frameworks and analysed facilities offered by them that allows us to create a simple simulation. Each of mentioned systems has its strengths and weaknesses in particular aspects of its usage in solving problems.

All of the systems have defined some kind of a life-cycle for agents. Some of them enforce it in a rather strict way (AgE, MadKit) and some provide very flexible mechanisms for scheduling an agent (RePast, MASON). Only RePast offers an easy way to schedule many methods from a single POJO class.

The second considered aspect was a possibility to organise agents in different ways. Both MASON and RePast have a good support for many (possibly co-existing in one simulation) organisational schemes (graphs, spatial neighbourhood, etc.). On the other hand MadKit requires following a strict Agent/Group/Role model but due to its flexibility one could possibly implement other arbitrary models with little effort. AgE also provides only one possible organisational model — a tree structure of agents.

The next aspect was communication and distribution. There is no defined communication schemes in RePast and MASON. The user has to rely on plain method calls or build their own mechanism. MadKit and AgE have chosen more strict approach and provide messaging facilities. Additionally, AgE has other means of communication available: queries and object properties. From described platforms only RePast provides an out-of-the-box way to distribute computation. In the rest of platforms user is required to come up with their own approach to this problem.

The last aspect was coupling. RePast and MASON both introduce a lot of interdependencies between agents (due to method calls as a messaging scheme) and with a platform (e.g. no built-in dependency injection). MadKit follows the scheme of agentification of services and also has rather low coupling between agents. AgE provides most advanced solutions regarding the decoupling and offers such facilities as external configuration, DI or easy runtime modifications.

5 Conclusions

In the course of the contribution, a review of selected Java-based, agent-oriented simulation frameworks was presented. The most important, in opinion of the authors, features of these programming environments were evaluated, such as agents life cycle management, inter-agent and agent-platform communication support, agents society organisation, distributed environment facilities, as well as the overall status of their development. The evaluation was conducted for the most popular general-purpose frameworks, such as MASON, RePast and MadKit, and proved that they had many advantages, such as overall maturity, sophisticated GUI or support for different scripting languages. However it was found that they all lacked either component-oriented features or built-in support for the low coupling of simulation structures. This leads to possibly inflexible designs, which makes it difficult to change isolated parts of the simulation system without altering their contextual dependencies.

Based on this scarce evaluation, the technical issues of AgE software environment were discussed. The platform seems to provide a well-suited execution environment for running agent-based simulations. It seems especially usefull when taking into consideration one of the most important issues of software engineering: flexibility and extensibility, which is achieved by the component-based design. Therefore it is very easy to extend the simulation prepared in AgE by changing its parts, parameters, upgrading them to the next versions etc.

The work on AgE development continues, in the near future implementation of user interface, monitoring facilities, experiment scheduling and persistence of results (among others) are planned to be completed.

References

1. Almasi, G., Gottlieb, A.: Highly Parallel Computing. Benjamin-Cummings publishers, Redwood City, CA (1989)
2. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall (2003)
3. Banks, J., Carson, J., Nelson, B., Nicol, D.: Discrete-Event System Simulation. Prentice Hall (2009)
4. Bellifemine, B., Poggi, A., Rimassa, G.: Jade – a fipa-compliant agent framework. In: Proc. of PAAM'99, London, pp. 97–108 (1999)
5. Bergenti, F., Gleizes, M.P., Zambonelli, F.: Methodologies and Software Engineering for Agent Systems. Kluwer Academic Publisher (2004)
6. Bhasker, J.: A SystemC Primer, Second Edition. Star Galaxy Publishing (2004)

7. Byrski, A., Kisiel-Dorohinicki, M.: Agent-based model and computing environment facilitating the development of distributed computational intelligence systems. In: Proc. of ICCS 2009. Springer LNCS 5544, 5545 (2009)
8. Byrski, A., Schaefer, R.: Stochastic model of evolutionary and immunological multi-agent systems: Mutually exclusive actions. Fundamenta Informaticae **95**(2-3), 263–285 (2009)
9. Carneiro, G., Fontes, H., Ricardo, M.: Fast prototyping of network protocols through ns-3 simulation model reuse. Simulation Modelling Practice and Theory **19**(9), 2063 – 2075 (2011)
10. Collier, N., North, M.: Repast SC++: A Platform for Large-scale Agent-based Modeling. Wiley (2011)
11. Dahmann, J.: High level architecture for simulation. In: Proc. of First International Workshop on Distributed Interactive Simulation and Real-Time Applications, pp. 9–14 (1997)
12. Davila, J., Uzcategui, M.: Galatea: A multi-agent simulation platform. modeling. In: Proc. of Simulation and Neural Networks (MSNN-2000) Merida, Venezuela, pp. 216–233 (2000)
13. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multiagents systems. In: Y. Demaseau (ed.) Proc. of ICMAS'98 Conference, Paris, pp. 128–135 (1998)
14. Fowler, M.: Inversion of control containers and the dependency injection pattern (2004). URL "http://martinfowler.com/articles/injection.html"
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
16. Gutknecht, O., Ferber, J.: The madkit agent platform architecture. In: In Agents Workshop on Infrastructure for Multi-Agent Systems, pp. 48–55 (2000)
17. Klein, J.: Breve: a 3d environment for the simulation of decentralized systems and articial life. In: Proc. of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems (2002)
18. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: A multi-agent simulation environment. Simulation: Transactions of the society for Modeling and Simulation International **82**(7), 517–527 (2005)
19. Nikolai, C., Madey, G.: Tools of the trade: A survey of various agent based modeling platforms. Journal of Artificial Societies and Social Simulation **12**(2) (2008)
20. North, M., Howe, T., Collier, N., Vos, J.: A declarative model assembly infrastructure for verification and validation. In: S. Takahashi, D. Sallach, J. Rouchier (eds.) Advancing Social Simulation: The First World Congress, Springer, Heidelberg, FRG (2007) (2007)
21. Pidd, M., Cassel, R.A.: Three phase simulation in java. In: In Proceedings of the 1998 Winter Simulation Conference, pp. 367–371 (1998)
22. Railsback, S., Lytinen, L.: Agent-based simulation platforms: review and development recommendations. Simulations **82**, 609–623 (2006)
23. Schaefer, R., Byrski, A., Smołka, M.: Stochastic model of evolutionary and immunological multi-agent systems: Parallel execution of local actions. Fundamenta Informaticae **95**(2-3), 325–348 (2009)
24. Stroock, D.: An Introduction to Markov Processes. Springer (2005)
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
26. Tesfatsion, L.: Agent-based computational economics: Modeling economies as complex adaptive systems. Information Sciences **149**(4), 262–268 (2003)
27. Ventroux, N., Guerre, A., Sassolas, T., Moutaoukil, L., Blanc, G., Bechara, C., David, R.: Sesam: An mpsoc simulation environment for dynamic application processing. In: CIT, pp. 1880–1886. IEEE Computer Society (2010)
28. Wainer, G., Mosterman, P.: Discrete-Event Modeling and Simulation: Theory and Applications (Computational Analysis, Synthesis, and Design of Dynamic Systems). CRC Press (2010)
29. Wooldridge, M., Jennings, N.: Intelligent agents: Theory and practice. Knowledge Engineering Review **10**(2) (1995)