Lightweight Distributed Component-oriented Multi-agent Simulation Platform

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

Abstract— **Existing solutions for agent-based systems turn out to be limited in some applications, like agent-based computing or simulations, where very large numbers of clearly defined agents interact heavily within a closed system. In those cases, fully-fledged, FIPA[1] compliant environment introduce unnecessary overhead, but simple tools fail to scale when confronted to bigger problems. In this paper, we introduce an alternative agent environment called *AgE*, targeted at medium-sized simulation and computational applications, which use multi-agent and computational intelligence paradigms, but does not need full FIPA compliancy, and would benefit from a component-based approach and distributed computing capabilities. After giving a short review of selected popular multi-agent platforms, the main features of AgE are presented. Next, some basic usability topics are addressed. Then the most interesting architectural aspects of the platform are discussed. Finally, AgE possibilities are demonstrated with two example applications.**

Keywords— **agent-based computing, component-based systems, agent-based simulation**

INTRODUCTION

Multi-agent systems (MAS), and more generally the concept of an intelligent agent, have found multiple applications, both as a mean to model complex systems and as a programming paradigm to implement them. Several established agent-based technological solutions exist, including FIPA compliant, fully-fledged environments like JADE or JADEX, where agents are a basic unit of software abstraction. As an example of another approach, there are also minimalistic and easy-to-use tools like NetLogo, where agents are only present at a domain level, as means of problem decomposition.

The first class of systems can be used to solve any problem that benefits from using an agent-oriented approach. However, in some particular classes of applications, this can be very inefficient. This happen especially in the case of simulation and computational applications, where agents and their behaviours are well defined. Such MAS might not need to be open to other systems, to require the possibility of code migration or to support FIPA-compliant communication.

In these cases, systems of the second class are more efficient and a much better choice. However, they suffer from other drawbacks, as they do not support component-oriented approach, which affects code reusability and make more complex problems hard to program. Also, these systems are usually not suited for running in distributed environments. In other words, they do not scale well with bigger problems or more complex systems.

In this paper, we introduce an alternative MAS environment called *AgE*, which aims to overcome the above issues. It is targeted at medium-sized simulation and computational applications, which use multi-agent and computational intelligence paradigms [1], but do not need full FIPA compliancy, and would benefit from a component-based approach and distributed computing capabilities. The presented work is a continuation of the platform presented in [2] and [3]. In this paper, however, we focus on component-orientation and distribution, as well as on the possibilities of the platform with regard to simulational and computational applications.

The AgE platform is designed with efficiency and usability in mind. We wanted to make it easy to experiment, prototype and tune simple MAS simulations or computations, then scale them to bigger problems and run in a distributed environment. In turn, the use of a component-based approach allows to create very flexible and reusable systems. It is important to stress that AgE is not meant as a replacement to existing agent-based technology, but rather as a convenient alternative dedicated to a specific class of applications.

The rest of this paper is structured as follows: First, a short review of some of the currently popular multi-agent systems is provided. Next, the main features of AgE from a multi-agent point of view are presented, including hierarchical environments, topologies and agent execution. After that, some basic usability topics are addressed, such as implementing custom agents or configuring applications using configuration mixins. Then, some of the more interesting architectural aspects of the platform are discussed, including agent execution, configuration and communication aspects. Finally, the advantages of using AgE are demonstrated with two example applications.

EXISTING SOLUTIONS FOR AGENT-BASED SYSTEMS

Agent-based software environments use agents as basic units of software abstraction. They focus on inter-agent relation and intra-agent intelligence [4], provide facilities for the discovery of agents, communication, life-cycle management etc. The FIPA standard allows to create such open, interoperable multi-agent systems, where fully-fledged autonomous software agents can express their needs or perceive the environment (and other agents) using specific languages, ontologies, etc. The most established solutions of this kind include JADE [5] and JADEX [6], the latter also using an active component approach.

A multi-agent system may also be implemented without any software structures corresponding to agents. This often happens in the case of simulations or computations, where

[1] Foundation for Intelligent Physical Agents

Proceedings 27th European Conference on Modelling and Simulation ©ECMS Webjørn Rekdalsbakken, Robin T. Bye, Houxiang Zhang (Editors)
ISBN: 978-0-9564944-6-7 / ISBN: 978-0-9564944-7-4 (CD)

the introduction of agents facilitates the modelling of complex phenomena, such as natural or social ones. In such cases, agents constitute building blocks of the *model*, which in turn may not at all be implemented with the use of agent technology. We describe briefly below some of the most popular tools of this kind, namely RePast [7], Mason [8], and MadKit [9].

MASON is an agent-oriented simulation framework developed at George Mason University. It is advertised as fast, portable, 100% Java based. The multi-layer architecture brings complete independence of the simulation logic from visualisation tools. The models are self-contained and may be included in other Java-based programs.

The programming model of MASON follows the 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. There are neither ready-to-use distributed computing facilities nor component-oriented solutions.

RePast is a widely used agent-based modelling and simulation tool. It has multiple implementations in several languages and built-in adaptive features such as genetic algorithms and regression. The framework uses fully concurrent discrete event scheduling. Dynamic access to the models in the runtime (introspection) is possible using a graphical user interface.

In Repast Simphony, a new organisational concept called 'context' was introduced. It consists in a group of unorganised agents (they may be organised 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 subcontext also exists in the parent context, but the reverse is usually not true.

MadKit is a modular and scalable multi-agent platform written in Java, aimed at modelling different agent organisations, groups and roles in artificial societies. It is built based on a so called Agent/Group/Role organisational model [10], using a plugin-based architecture. The architecture of MadKit is based on micro-kernels which provide only the basic facilities: local messaging, 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 and scheduled agents may be developed.

Simulations do not require any particular structure or model to run. However, it is possible to add an arbitrary scheduler or create complex agent communities and relationships. Agents can locate other agents having some specific role or belonging to some particular group. Agents can communicate with each other using these roles or group membership (i.e. using broadcast messages).

MAS FEATURES

The following section describes the main features of the AgE platform from a MAS point of view, as they are available to end users.

A. Agent Types

The AgE platform includes two types of agents: heavyweight agents and lightweight ones.

Heavy agents are realised as separate threads, as in the JADE platform. They communicate through asynchronous message passing. This is an effective model when the number of agents is small or agent interactions are sparse (i.e. each agents communicates only with a small number of other agents).

However, in simulation and computation scenarios where there is a large number of heavily communicating agents, experiments have shown that a fully concurrent approach does not perform efficiently. In fact, most of the execution time tends to be spend on handling message queues and switching threads context[11].

In order to overcome this issue, AgE introduces *lightweight agents*. Populations of such agents are thread-contained and execute pseudo-concurrently, one step each at a time. Some restrictions on how agents may change each other state, along with communication constraints, described further below, emulate concurrency effects and execution interleaving. From these agents perspective, they are effectively processed in parallel.

Thus, the platform API available to heavyweight and lightweight agents is very similar. The main difference is in efficiency and determinism, as needed in a particular case.

B. Agent Environments

Each agent in AgE belongs to some environment. Agents can communicate with each other within their environment or query it to acquire some information.

In the case of heavy agents, the environment simply consists of multiple distributed nodes. All heavy agents on all connected nodes belongs to it. For more information on this virtual execution environment, see Section -F.

B.1 Hierarchy of Environments

When it comes to lightweight agents, environments are treated as agents themselves. These *aggregate* lightweight agents also have properties, behaviour, and also execute in some higher level environment. Thus, following the Composite design pattern [12], lightweight agents form hierarchies, as shown in Fig. 1. The root agent is a heavy one and is called a *workspace*. A workspace encompasses all the hierarchy within its thread and initiates step based, pseudo-concurrent execution.

As an application of agents hierarchies, consider a flocking MAS where multiple flocks, composed of multiple agents, coexist. Particular agents can interact within a flock, but flocks as wholes can also interact. They can compete over limited resources or exchange information, reflected by the agents within them.

B.2 Topology of Environments

By default, all environments are assumed flat, i.e. all contained agents can see each other. It is however possible for users to restrict this visibility, by defining a custom topology.

As an example, consider a topology which takes into account the network latency between distributed computing

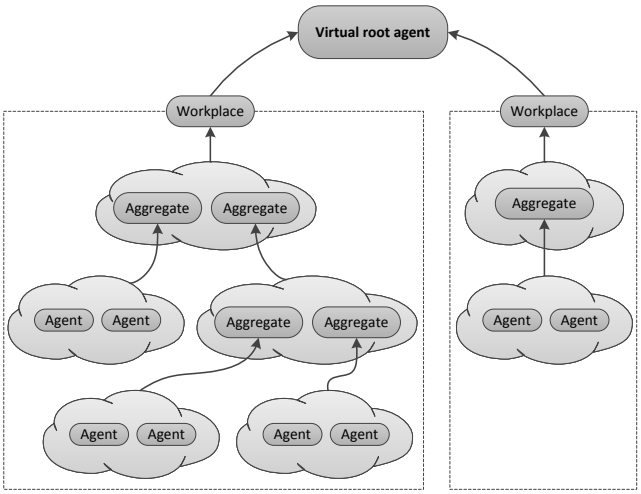

Fig. 1: Lightweight agents environments can form hierarchies. Each environment agent is called an aggregate. The root one is called a workspace and is implemented as a heavy agent. The whole hierarchy executes within the root's thread.

nodes. Limiting interactions to agents physically nearby can dramatically increase performance. Another example would be a topology which clusters agents based on their features, then restricts visibility to within a cluster.

Remark that introducing a topology is not the same as splitting an environment into multiple ones, as the induced relation can be arbitrary. In particular, it needs not be symmetric or transitive.

C. Agent Execution

As mentioned before, heavy agents execute concurrently in their own threads, usually spinning in a loop or waiting for incoming messages.

Lightweight agents in turn execute pseudo-concurrently, one step at a time. The order of execution in a given environment is by default arbitrary (though this policy can be customized). Child agents in the hierarchy execute their step before their parent, so that changes bubble up the tree from leaves to the root.

A consistent iteration order over the agent hierarchy needs to be preserved. To this end, any structural modifications of an environment (such as the addition, removal or migration of agents) is postponed until all the agents in that environment are done with their step. In consequence, any such modification will only be visible in the environment in the next step.

Beside executing custom logic, both heavyweight and lightweight agents can interact with each other through asynchronous *messages* or perform *queries* on their environment. Additionally, lightweight agents are free to interact synchronously within a given environment and to submit asynchronous *actions* to their environment for later execution.

C.1 Messages

Agent communication in AgE is mainly based on message-passing. Immutable messages consist in an target address and arbitrary payload.

All agents are provided with an unique address by a distributed addressing service. Messages can be targeted for unicast, multicast, anycast or broadcast transmission.

Agents do not send messages directly. Instead, they submit them to the environment, which handles delivery. Heavy agent messages are handled fully concurrently. In the case of lightweight agents, however, a message sent in one step will be received in the next step.

C.2 Queries

Queries offer agents the possibility to gather, analyse and process data from both local and remote environments. They allow to perform tasks like computing the average value of some agent property in the environment, select and inspect arbitrary objects in collections and much more. They can be applied to arbitrary properties using reflection or specialized for efficiency.

Queries are defined using a simple, declarative, yet extensible query language, following the Fluent Interface design pattern [12]. They are built without the knowledge of the target object, so a single query can be reused on multiple objects, yielding different results.

The platform also offers the possibility to cache query results for efficiency. Such a wrapping query is given some expiration time and will recompute the result only if that time has elapsed.

C.3 Actions

Sometimes step-based processing does not provide appropriate granularity. In particular, a single step may be composed of several distinct phases which should execute consecutively.

As an example, consider an algorithm where, in each step, agents first exchange energy, then reproduce and create new agents. Obviously, we would like that all agents first exchange energy, then all agents potentially reproduce. What we get instead, with simple step processing, is that the first agent exchanges energy and reproduces, then the second agent does the same, etc.

In order to achieve truly *interleaved execution* with lightweight agents, the AgE platform introduces asynchronous *actions*. Following the Command Object design pattern [12], instead of executing at once, parts of the algorithm are encapsulated into actions objects. These actions are then submitted to the environment for later execution.

When all agents in the environment have finished their step, the environment executes the submitted actions. They are first reordered so that actions of the same type are grouped together. Then, such groups are executed in a FIFO order (this policy can also be customized, by providing an explicit total order over action types).

Using this method, a single step can be decomposed into a sequence of actions, which are effectively interleaved.

As the actions to be executed can be injected into the agent from configuration, this lead to an interesting model: Agents types only define state, but behaviour is fully encapsulated into actions and composed at configuration time. In order to change the behaviour of an agent, only the configuration needs to be updated, without recompilation. Multi-

ple similar agents can also have slightly different behaviour. Moreover, it also becomes possible to change the behaviour at runtime.

USING THE PLATFORM

This section briefly describes MAS simulations and computations can be implemented and configured in AgE.

The platform has been intentionally designed to use standard Java technologies, concepts and patterns, so that it could be used just like other mainstream MAS or component platforms. Because of that, we believe it to have an efficient learning curve.

D. Creating Agents

Agents in AgE are implemented in Java or Scala with the use of a very simple API, reflecting the features described in Sec. . The platform then handles distribution and concurrency transparently to the user.

Just like in Jadex, agents are actually components managed by an IoC container. They can have properties and dependencies, the latter being wired and injected by the container.

Specific parts of the algorithm, such as a clustering method or a preselection technique in a genetic algorithm, can be extracted to dedicated components. These can be shared, as dependencies, by multiple agents, following the Strategy design pattern. The platform provides several such components out of the box, especially for evolutionary computations.

E. Configuring an Application

The configuration of an AgE application is decoupled from the implementation of agents and other components. The main way to describe an AgE application is to provide an XML configuration file with a syntax very similar to the one found in frameworks like Spring.

In such a file, the user can define all of the components, initialize their properties, specify explicit dependencies (optionally, as these will be autowired otherwise). This is all very similar to Spring configurations.

However, there are a few notable outstanding features when compared to Spring or similar solutions. These features are *nested definitions* and *configuration mixins*.

In AgE, component definitions can be nested in one another. This is different from inner beans found in Spring, as inner components will only be visible to the outer component and other components nested inside it. Such a solution allows to solve the so-called *robot-leg problem*.

E.1 Configuration mixins

The other novel feature is the possibility of configuration *mixins*. Most frameworks allow users to include configuration files one in another. AgE takes this one step further: it is possible to define named structural blocks in configuration files. When including some other file, the blocks it contains can then be extended or overridden.

In this way, one can achieve inheritance-like functionality, or more generally — mixins (as inclusion can happen in multiple places).

This greatly improves the reusability of configuration files, as it becomes possible to create complex, but general configurations. Custom computations need only define components specific to the problem being solved.

As an example, consider the configuration for an evolutionary algorithm, where operators such as selection or variation are given default implementations. Such a configuration can be reused as is — the user just needs to include it and provide some representation and fitness function. However, the user can also overwrite particular blocks in the original configuration, in order to supply custom operators.

IMPLEMENTATION ASPECTS

In this section we will briefly discuss the most important technical aspects of the platform.

F. Virtual Execution Environment

The platform introduces a *virtual execution environment* (Fig. 2). In distributed systems, it allows to perform operations involving workplaces located on different nodes, without their awareness of physical distribution. Such operations are executed by the core service according to the *Proxy* design pattern [12]. A core service uses the communication service to communicate with core services located on other nodes.

The global namespace of agents can be narrowed by introducing an *agent neighbourhood*, that defines the visibility of top level agents. An agent can only interact with agents in its neighbourhood. For heavy agents, the neighbourhood is realised and managed by a topology service. In the case of lightweight agents, this functionality is performed by the environment, which will usually delegate it to some dedicated component.

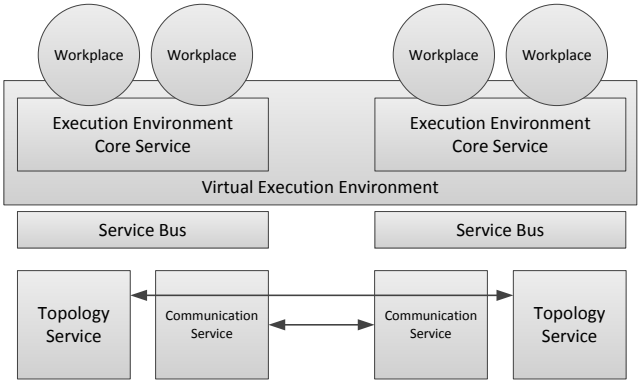

Fig. 2: Virtual execution environment. Workplaces see a unified, global namespace of agents and are not aware of physical distribution among many nodes. Communication is carried by the communication service. The topology service may introduce logical divisions of the namespace.

G. Agents Execution

Lightweight agents execute pseudo-concurrently, by repeatedly executing some step callback function. From their point of view, lightweight agents effectively execute in parallel. The platform emulates this parallelism by introducing

several constrains on message propagation and change visibility.

First, messages have a positive time of propagation. It means that a message send in one step will only be received at the earliest in the next step. Lightweight agents must thus assume latency just like heavy ones.

Secondly, within a given environment, lightweight agents can interact synchronously. Obviously, there is no need for synchronization, as they actually execute sequentially. However, there is no guarantee (from an agent point of view) over the order in which they do. Therefore, they must make assumptions similar to those they would need in a truly concurrent environment.

Thirdly, the platform enforces some strict semantics on the way changes propagate in the environments hierarchy. In particular, agents in a given environment are assumed parallel, but agents in separate environments are assumed independent.

This implicates that any agent in the hierarchy can only directly (synchronously) change a sibling or child agent (if it is itself an aggregate). However, all the agents above in the tree are available in "read only" mode. This which ensures that agents in two different subtrees can never make conflicting changes.

Of course, it may be necessary to modify the agent tree above, for example by adding a new agent to the environment. Some changes can be requested simply a sending a message to the target agents. Changes which need to be executed before the next step (such as adding a new agent to the environment) can be wrapped in asynchronous actions submitted to the environment.

An aggregate agent will first let its children carry out their tasks. Then, it will execute the actions which they submitted. Finally, it will execute its own tasks.

Therefore, from the point of view of the whole agent tree, actual execution is performed in a post-order way, from the leaves up to the root, level after level. With each level, the visibility restriction are softened, and eventually, all leaf environments are synchronized in the root.

G.1 Messages

Message passing (described in Sec. -C.1) is the basic communication mechanism offered by AgE, simple but flexible. Both agents and services use it. However, types of a sender and receivers cannot be mixed — there is no possibility for an agent to send a message to a service and vice versa. They can be easily extended to use more structured payloads and carry additional metadata in headers.

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

As mentioned above, in order to emulate asynchronous messaging, a message sent by an lightweight agent in a given step will be available to the receiver in the next step.

H. Component Model and Framework

The platform defines *components* close to classes, i.e. a component is represented by a single class (called *component stub*) enhanced with additional description (called *component specification*). Such an approach gives a possibility to construct computational agent-based systems from relatively small parts and gain advantages from component-oriented programming. However classes are not a good unit of deployment [13, chap. 2]. Therefore the deployment model assumes that component classes are provisioned and deployed in bigger containers, such as jar files in the Java environment.

A component in the runtime environment is identified by a unique textual name, which is generated by default from a fully qualified name of the component stub. A component specifies its *capabilities*, *requirements* and *configurable properties*.

By default all interfaces implemented by the class are treated as component capabilities, but there is a possibility to narrow them by explicit specification using Java annotations.

Requirements specify conditions that must be fulfilled before a component can serve its capabilities. The presented model defines a requirement as the declaration of a dependency to some other interface. Dependencies can consist in annotated fields, methods or constructor parameters, following the JSR-330 specification. A dependency declaration can be enhanced with an additional qualifier which points to a concrete name of a component. Such dependencies are called *named dependencies*. Configurable properties are defined according to the Java Beans convention.

The model does not specify one concrete method of defining a component specification – particular implementations can use different methods.

The platform uses the dependency injection pattern and is built on top of the PicoContainer framework[2]. A container is used to instantiate and assembly component instances, based on a system configuration. A configuration can come in a variety of formats and syntaxes: it could be an architecture description language expressed in XML format or written in a dedicated Domain Specific Language, or be defined in some interactive console or GUI. All these must be interpreted in a consistent way by an appropriate configuration subsystem.

In order to express this diversity, the architecture of such a subsystem could become excessively complex. To avoid that, the representation of the configuration has been split into two layers: a user-specific format, and a Configuration Object Model.

As shown in Figure 3, multiple user format and tools are acceptable (the default, XML-based approach was discussed in Section -E). They are transformed into a common abstract representation, the Configuration Object Model, which can be used in turn by an IoC Container to instantiate a concrete graph of component instances.

This approach decouples the way the configuration is seen by the users from the way it is interpreted by the system. Adding a new configuration format is just a matter

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

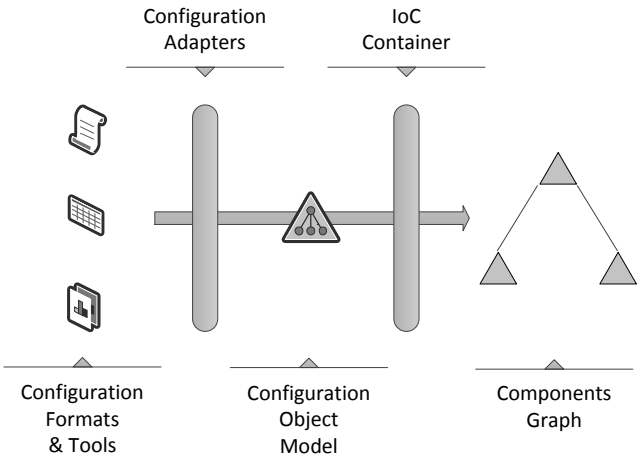

Fig. 3: Multiple configuration adapters may handle different configuration formats, translating them to a common Configuration Object Model. This common representation is in turn used by the container to instantiate a graph of components.

of creating a corresponding adapter, while the in-memory representation of the configuration can be kept as compact and simple as possible. Both can then evolve independently, making the system open for extensions.

H.1 Configuration Object Model

The *configuration object model* is an abstract way to describe a graph of components. It contains all information required to instantiate and initialise components and wire them up together.

The configuration object model is a collection of component definitions that represent component runtime configurations defined in the presented component model. These consist of a unique name, a type and a scope. The scope of a component definition can be either a *singleton* or a *prototype* one.

A definition's *singleton* scope indicates that there should exist at most one instance of the component it describes. If the definition has a *prototype* scope, it indicates that each dependency to this component should be resolved with a separate instance, following the Prototype pattern [12].

Definitions can be assigned constructor and property arguments, which are used to instantiate and initialise components. These arguments can be a reference to another definition (i.e. a dependency to other component expressed by a name of the target), or a value-type parsable from some string representation (i.e. a configurable property).

Component definitions can be nested and effectively form a tree. Inner definitions are only visible to their parent and its other descendants, which form together a *context*. When resolving dependencies, components from the same context take precedence over those from the outside one. This makes it possible to create hierarchical configurations, where inner definitions can hide outer ones by type or by name (this is the only exception for the name-uniqueness).

Together with *mixins* in XML files, this feature improves configuration reusability, as global configuration files can define components for global or mock services, but these can be hidden by local implementations.

H.2 Components Initialisation

A component can be instantiated when all of its requirements are met. An object of the component stub class is created, all dependencies are initialised with references to instances of other components (created if needed) and properties values are set to their defaults. After that, depending on component attributes, some additional initialisation operations can be performed. Components can be instantiated at system startup (eager loading) or on demand, before first use (lazy loading).

H.3 Relationship Between Components and Agents

Some agents may perform similar tasks, but work with different structures and mechanisms. Thus, from a software engineering perspective, it may be said that the system is decomposed into particular agents, but a single agent implementation is too complex to serve as an assembly unit.

Agents implementations may actually be further decomposed into functional parts (components), which are replaceable, as long as they are compatible to one another, even when used by different agents (this ensures agents interoperability at implementation level). This is a realization of the Strategy design pattern [12].

A particular application is thus decomposed into agents and strategies, which provide the implementation of some specific part of agent behaviour. Both agents and strategies are implemented as classes and they are provided to a runtime environment as components. But, in regard of their different characteristic, they are provided with two distinct scopes.

Agents are represented as objects with well-defined lifecycle and attributes which define their state. Many agents of the same type (i.e. the same class) can exist in one environment. Moreover some of them can be differently configured: have different values of named properties or various implementations of dependent strategies.

According to such assumptions, agents are provided as components with a *prototype* scope. This allow to create many instances of the same agent type with the same configuration. However, these instances are not identifiable. To solve this issue, agent addresses have been introduced. Agents of the same type, but configured differently, are provided as distinct components with different names, however pointing to the same class.

In turn, strategies are provided to the environment as components with a *singleton* scope. It is sufficient to hold only one instance of a strategy, as it provides only stateless action implementations. Therefore, strategies can be identified by components names and there is no need to introduce an additional address schema.

I. Distributed Computation

The platform assumes that a computation may be executed in a distributed environment, comprised of nodes connected via communication services. Each node is a separated process, possibly executed on a different physical machine. This means that two modes of distribution are distinguished: on the service level (opaque) and on the computation level (transparent). This simply means that services in a given node are aware of the existence of other nodes

and know that to communicate with them, they need to use a specialised facility. On the other hand, agents should not notice that their communication is carried over the network.

Services use the communication service by simple method calling. Usually, communication between remote services is limited to services of the same type and performed using a message passing with the platform-wide message types. However, messaging is not the only way for services to communicate. Currently, basic synchronisation primitives, like global (environment-wide) barriers, are also available.

The separation between the underlying deployment scenario and the computation itself gives us even more possibilities. An amount of work may differ a lot between workplaces. Moreover, workplaces can be moved between nodes. These two facts lead straightforwardly to implementing load balancing on a workplace level. Workplaces can be migrated from more loaded nodes to these that are free. This also makes it possible to add more nodes to an already running computation to speed it up.

Computations may also be run one after another, on the same node instances. After finishing one execution, another configuration file may be loaded and a new computation deployed.

APPLICATION EXAMPLES

J. Transport Simulation

We have successfully implemented and tested several transport simulation examples. We will briefly present one of them as a demonstration of how the concept of agent hierarchy can be used.

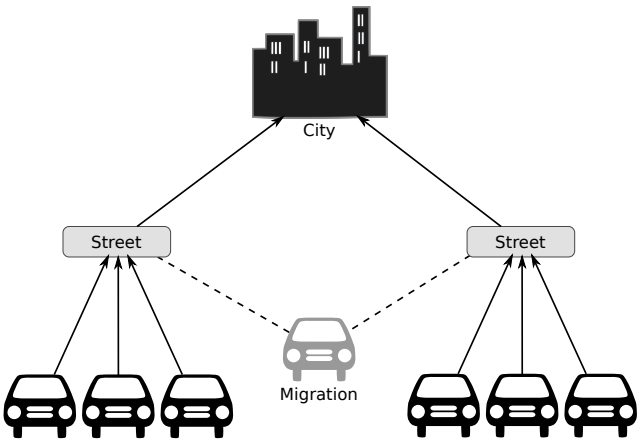

Fig. 4: Hierarchy of agents in a transport simulation example. The top-level aggregate represents a city. The next level consists in street aggregates. Simple agents on the lowest level represent individual cars.

This scenario consists of simulating road transport in a city with continuous space. We distinguish three main components interacting with each other: cars (or any other road vehicles), streets and an environment. All of their instances are represented as agents. This model is simple but general could easily be extended by introducing further types of objects into the simulation: pedestrians, bicycles, static obstacles, etc.

Cars are implemented as simple, lightweight agents, collected in streets which are represented as aggregates. Similarly, all streets are collected in the aggregate which represents the whole city (or a district if its large enough).

Each car can have a separate and unique driving strategy. Car movements within a street are efficiently computed within the aggregate, by executing agent actions such as braking, turning, etc. Cars movements between streets are represented as agents migrations. Collisions and illegal movements are easily detected by street aggregates.

K. Evolutionary Multi-agent Systems

Another area of application of lightweight agents and agent hierarchies is evolutionary optimization. Multi-agent systems can be combined with evolutionary algorithms, in order to solve optimization problems[14][15].

Individual agents are assigned with a solution to the problem and some coresponding fitness. They are also given some initial energy and then put to competition. Agents with a higher fitness take energy from those with lower fitness, inducing an emergent selective pressure.

Agents die if their energy drops to zero. In turn, if they gather enough energy, they can reproduce, yielding new agents. These are assigned with new solutions, obtained through standard evolutionary operators applied on the parents.

Competition can also happen on a higher level, by grouping agents into competing flocks[16], resulting in a more general version of the Island Model in Evolutionary Computation.

Arbitrarily nested flocks can easily be represented in AgE agent hierarchy. Individual agent interactions, such as fights and reproduction, can be represented as actions. Agent in different flocks can in turn have separate evolutionary operators, also allowing meta-optimization of the algorithm.

CONCLUSIONS

In this paper, we have presented an MAS computing environment called AgE, targeted at medium-sized simulation and computational applications, which use multi-agent and computational intelligence paradigms, but does not need full FIPA compliancy, and would benefit from a component-based approach and distributed computing capabilities.

Up till now, we have been able to successfully implement and test various simulation scenarios. We consider two of them as the most interesting: transport simulation [17] and evolutionary multi-agent systems [18]. The former extensively used both the hierarchy capabilities of AgE and its configuration and re-use facilities. The latter is an idea based on joining the classical evolutionary computation with agent-based paradigms and is an unusual approach e.g. to artificial life.

Through the concept of lightweight agents, AgE is optimized for a large numbers of heavily interacting agents, which corresponds to many agent-based simulation and computational applications. The platform can also be distributed over multiple nodes. A component-oriented approach allows to achieve high flexibility and reusability,

both at code and at configuration level because of configuration mixins. The use of technological standards results in low switching costs from other agent technology and an efficient learning curve. As such, we believe the AgE platform to be a convenient alternative, dedicated to a specific class of applications.

The work on the development of AgE continues. In particular, we consider the following improvements in the near future:

- stronger support for the Scala language, along with a functional agent paradigm and actor-based concurrency,
- visualization facilities, experiment scheduling and persistence of results,
- component migration, dynamic reconfiguration, acquisition of components from remote repositories,
- virtualised deployment in the cloud.

ACKNOWLEDGMENT

The research presented here was partially supported by the grant "Biologically inspired mechanisms in planning and management of dynamic environments" funded by the Polish National Science Centre, No. N N516 500039.

REFERENCES

[1] M. Kisiel-Dorohinicki, G. Dobrowolski, and E. Nawarecki, "Agent populations as computational intelligence," in *Neural Networks and Soft Computing*, ser. Advances in Soft Computing, L. Rutkowski and J. Kacprzyk, Eds. Physica-Verlag, 2003, pp. 608–613.

[2] A. Byrski and M. Kisiel-Dorohinicki, "Agent-based model and computing environment facilitating the development of distributed computational intelligence systems," in *Computational Science - ICCS 2009, 9th International Conference*, ser. LNCS, vol. 5544. Springer-Verlag, 2009.

[3] Ł. Faber, K. Pietak, A. Byrski, and M. Kisiel-Dorohinicki, "Agent-based simulation in AgE framework," in *Advances in Intelligent Modelling and Simulation*, ser. Studies in Computational Intelligence, A. Byrski, Z. Oplatková, M. Carvalho, and M. Kisiel-Dorohinicki, Eds. Springer Berlin Heidelberg, 2012, vol. 416, pp. 55–83.

[4] M. Wooldridge, *Reasoning About Rational Agents*. The MIT Press, 2000.

[5] B. Bellifemine, A. Poggi, and G. Rimassa, "Jade – a fipa-compliant agent framework," in *Proc. of PAAM'99, London*, 1999, pp. 97–108.

[6] A. Pokahr, L. Braubach, and W. Lamersdorf, "Jadex: Implementing a bdi-infrastructure for jade agents," *EXP - in search of innovation (Special Issue on JADE)*, vol. 3, no. 3, pp. 76–85, 9 2003.

[7] M. North, T. Howe, N. Collier, and J. Vos, "A declarative model assembly infrastructure for verification and validation," in *Advancing Social Simulation: The First World Congress, Springer, Heidelberg, FRG (2007)*, S. Takahashi, D. Sallach, and J. Rouchier, Eds., 2007.

[8] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A multi-agent simulation environment," *Simulation: Transactions of the society for Modeling and Simulation International*, vol. 82, no. 7, pp. 517–527, 2005.

[9] O. Gutknecht and J. Ferber, "The madkit agent platform architecture," in *In Agents Workshop on Infrastructure for Multi-Agent Systems*, 2000, pp. 48–55.

[10] J. Ferber and O. Gutknecht, "A meta-model for the analysis and design of organizations in multiagents systems," in *Proc. of ICMAS'98 Conference, Paris*, Y. Demaseau, Ed., 1998, pp. 128–135.

[11] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," http://disruptor.googlecode.com/files/Disruptor-1.0.pdf, 2011, [Online; last access 31-March-2013].

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] A. J. A. Wang and K. Qian, *Component-Oriented Programming*. Wiley-Interscience, 2005.

[14] K. Cetnarowicz, M. Kisiel-Dorohinicki, and E. Nawarecki, "The application of evolution process in multi-agent world to the prediction system," in *Proceedings of the Second International Conference on Multi-Agent Systems, ICMAS*, vol. 96, 1996, pp. 26–32.

[15] A. Byrski and M. Kisiel-Dorohinicki, "Agent-based evolutionary and immunological optimization," in *Computational Science–ICCS 2007*. Springer, 2007, pp. 928–935.

[16] M. Kisiel-Dorohinicki, "Flock-based architecture for distributed evolutionary algorithms," *Artificial Intelligence and Soft Computing-ICAISC 2004*, pp. 841–846, 2004.

[17] E. Nawarecki, J. Koźlak, G. Dobrowolski, and M. Kisiel-Dorohinicki, "Discovery of crises via agent-based simulation of a transportation system," in *Multi-Agent Systems and Applications IV*, ser. LNAI, M. Pěchouček, P. Petta, and L. Varga, Eds., vol. 3690. Springer-Verlag, 2005, pp. 132–141.

[18] M. Kisiel-Dorohinicki, "Agent-based models and platforms for parallel evolutionary algorithms," in *Computational Science – ICCS 2004. Part III*, ser. LNAI, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., vol. 3038. Springer-Verlag, 2004, pp. 225–236.

ABOUT THE AUTHORS

Daniel Krzywicki obtained his M.Sc. in 2012 at AGH University of Science and Technology in Cracow and is currently a Ph.D. student at the Department of Computer Science of AGH-UST. His research interests include agent-based computations, functional programming and distributed systems.

Łukasz Faber obtained his M.Sc. in 2012 at AGH University of Science and Technology in Cracow and is currently a Ph.D. student at the Department of Computer Science of AGH-UST. His research interests include agent-based modeling and distributed systems.

Kamil Piętak obtained M.Sc. in 2008 at AGH University of Science and Technology in Cracow and is currently a Ph.D. student at the Department of Computer Science of AGH-UST. His research interests include component-oriented programming and Eclipse RCP technologies.

Aleksander Byrski obtained his Ph.D. in 2007 at AGH University of Science and Technology in Cracow. He works as an assistant professor at the Department of Computer Science of AGH-UST. His research focuses on multi-agent systems, biologically-inspired computing and other soft computing methods.

Marek Kisiel-Dorohinicki obtained his Ph.D. in 2001 at AGH University of Science and Technology in Cracow. He works as an assistant professor at the Department of Computer Science of AGH-UST. His research focuses on intelligent software systems, particularly using agent technology and evolutionary algorithms, but also other soft computing techniques.