INFORMATICA, 2024, Vol. 35, No. 2, 379–400

DOI: https://doi.org/10.15388/24-INFOR555

Three New Design Patterns for Scalable Agent-Based Computing and Simulation

Mateusz NAJDEK, Mateusz PACIOREK, Wojciech TUREK, Aleksander BYRSKI*

AGH University of Krakow, Al. Mickiewicza 30, 30-059 Krakow, Poland
e-mail: najdek@agh.edu.pl, mpaciorek@agh.edu.pl, wojciech.turek@agh.edu.pl, olekb@agh.edu.pl

Received: October 2022; accepted: April 2024

Abstract. Multi-agent approach is very popular for modelling and simulation of complex phenomena, design and programming of decentralised computing systems. Asynchronous beings, which do not share state but communicate using messages are a convenient abstraction for representing various phenomena observed in the real world. When the number of the considered agents grows, the designers and developers of such systems must address the problem of performance. Introducing distribution is often a weapon of choice, which, however, does not guarantee obtaining proper scalability and efficiency. The intensity of communication in a large-scale agent-based system can easily exceed the abilities of a distributed hardware architecture, leading to poor performance. After analysing various distributed agent-based systems, we identified several reasons for limited performance and several architectural solutions, which can help overcoming this problem. The main aim of the presented work is identification and systematization of these architectural solutions in the form of design patterns. As a result, we propose three new design patterns for building scalable distributed agent-based systems. A systematic description of their aims, structure, variants and features is provided, together with examples of applications.

Key words: design patterns, multi-agent systems, agent-based simulation, agent-based computing, scalability, distribution.

1. Introduction

Similarly to many researchers working in computer science, we have been observing the development of the agent paradigm for more than 20 years. From the very beginning of our research, we applied its concepts and abstractions for solving many different problems, including computing (Krzywicki *et al.*, 2015), simulation (Boronea *et al.*, 2009), robot management (Turek *et al.*, 2011) and many others. Together with other researchers involved in the community, we tried to adopt and extend specifications, use a variety of platforms, get inspired by agent-based solutions created worldwide, design and create our own frameworks and systems (Turek *et al.*, 2016; Bujas *et al.*, 2019). We came to the point

*Corresponding author.

where we have to admit that we are amazed by the capacity of the seemingly consistent concept of the software agent.

While investigating numerous cases of using the agent paradigm, we came across several surprisingly scalable and efficient computing systems. The motivation for writing this paper comes from interesting observations drawn from their analysis. These systems, originally created for different purposes by different teams and at different times, share a common need: the need for high computing efficiency. While considering completely different computational tasks, the system profit from using multi-core and multi-node computing infrastructures and execute multi-agent system in a distributed manner. Their features allow for processing large tasks or large numbers of tasks in a reasonable time and for achieving proper scalability on distributed hardware.

The analysis of several exemplary agent-based systems, which will be presented in detail in this paper, reveals obvious similarities in the designed architectures of the multi-agent systems. Reinventing architectural concepts in different systems by different teams suggests the existence of common problems and common solutions to these problems. Such a general solution is typically called a *design pattern* in software engineering.

Identification and systematization of design patterns for providing efficiency and scalability of large-scale multi-agent computing and simulation systems is the main aim of this work. As a result, we propose three new design patters for solving common efficiency and scalability problems, which appear in many agent-based systems. The patterns allow for managing numerous populations of agents (the Colony pattern), for performing interactions between agents concurrently (the Arena pattern) and for distributing the populations of agents on many computing nodes (the Breeders pattern).

In the next section, a review of existing work on design patterns for agent-based systems will be presented. The following section will present the architecture of three different multi-agent systems, which demonstrated good scalability of computations. Later, three common architectural concepts are abstracted from the systems and described in the form of design patterns. The cooperation and characteristics of the patterns are presented in Section 5, which is followed by the Conclusions.

2. Design Patterns for Agent-Based Systems

A design pattern is an abstract solution for solving a common problem. It has at least two major advantages: acceleration of the design and development process and standardization of the created solutions. A good practice is to identify possible design patterns applications at a very early stage of the problem analysis and throughout the whole development process. This approach imposes a standardized structure of code which would significantly simplify its extensions and corrections.

Design patterns are very important elements of every programming paradigm. In object-oriented paradigm the design patterns proposed by the Gang of Four in Gamma *et al.* (1995) formulate the common basis for solving the majority of architectural and design problems. The basic set of 23 patterns has been extended over the years with

many new concepts (Asghar *et al.*, 2019). Other programming paradigms also benefit from proven solutions for common problems. In aspect-oriented paradigm, the common problems can be addressed with solutions inspired by the patterns proposed by the Gang of Four (Vaira and Čaplinskas, 2011). In functional paradigm, a set of functional design patterns for applying actions onto collections of data is a commonly used approach (Lämmel and Visser, 2002). In the case of the agent-oriented programming paradigm, the process of identification and standardization of the most important design patterns is ongoing – it seems that we are still waiting for the "Gang of Four for Agents".

The first attempts to define design patterns in the cortex of software agents can be found in the work of Aridor and Lange from 1998 (Aridor and Lange, 1998). The work proposes a simple taxonomy for typical tasks of *mobile agents*, which is a somewhat forgotten concept of software pieces that can travel between computers in a network. Over the next decades, researchers have proposed a wide variety of solutions to particular problems identified during agent-based systems design, which cover a far more general scope of possible software agent shapes and applications. Examples can be found in Klügl and Karlsson (2009), where a set of design patterns for the problems related to agent-based simulation models is presented. The authors adopt the pattern description method proposed by Gamma *et al.* (1995) and define several design patterns for internal architecture of an agent, agent population, interactions between agents and environment representation. The work does not provide formalization or examples of using the patterns in verified systems. Another attempt to defining design patterns for agent-based simulations is presented in Barrera *et al.* (2011). The authors try to express the considered design problems with the Unified Modelling Language (UML), which has been designed within the object oriented paradigm. The proposed patters, related to agents behaviours, are applied in a single case study. The process of identifying patterns in existing systems has been presented in Mathieu *et al.* (2018) and Sicard *et al.* (2021). The authors demonstrate examples of concrete problems emerging in agent-based simulations and propose generic solutions to these problems. The design patterns, which focus on organization of multi-level agent-based architectures, are described using several different means, including description, ULM diagrams and code of algorithms. Conceptual model for patterns and agent-based abstraction is given in Mohamad *et al.* (2007).

Several attempts to the challenge of systematization of agent-based design patterns can be found in the works of Lind (2003), Weiss (2003), Hofstede and Chappin (2021) and Sauvage (2004), where design patterns description format is proposed, and in Oluyomi *et al.* (2007), where the authors describe and classify 28 patterns. North and Macal (2011) present a survey of the agent-based patterns which were actually applied several times by other authors. Probably the most comprehensive review of patterns can be found in Juziuk *et al.* (2014) where a systematic method of finding and classifying is proposed. The work identifies 206 patterns and analyses their structure, description, and possible range of applications. Interestingly, the authors point out the aspects of cloud and grid computing as the ones which did not receive sufficient attention in this context. The design patterns for scalable multi-agent systems, presented in this paper, aim to fill this gap, keeping in mind comments such as the ones which can be found in e.g. Cruz Torres *et al.* (2011), showing

design patterns as one of feasible ways for full implementation of agent-based systems in real-world applications.

3. Scalable Multi-Agent Systems: Case Studies

The three selected multi-agent systems, which will be used as examples for successful scalability of an agent-based implementation, were designed and developed for solving three different computing problems:

- optimization problems solved with evolutionary metaheuristics,
- autonomous being simulation in complex spaces,
- network traffic control and planning.

Despite obvious differences in the considered areas, significant similarities can be observed in the agent-based architectures presented below.

3.1. *ParaPhrase EMAS: Evolutionary Agent-Based Metaheuristics*

Combining the agent-oriented approach with population-based metaheuristics (usually employing large numbers of entities) seems natural. Let us keep in mind that computing metaheuristics usually consist of a large number of individuals (calling for implementation means involving concurrency and distribution), and implementing such systems based on existing agent-based approaches, frameworks, etc. (see e.g. JADE) will be perhaps flexible, but not scalable. Indeed, scalability and the possibility to efficiently use supercomputing may be treated here as a feature that cannot be overestimated, for achieving a well-behaved metaheuristic implementation.

Let us focus for a moment on a selected metaheuristic algorithm, namely Evolutionary Multi-Agent System (EMAS) proposed in 1996 (Cetnarowicz *et al.*, 1996) and since then developed further (for a good survey on EMAS refer to Byrski *et al.*, 2015).

Each agent is assigned with a real-valued vector representing a potential solution to the optimization problem, along with the corresponding fitness.

Emergent selective pressure is achieved by giving agents a single non-renewable resource called energy (Cetnarowicz *et al.*, 1996). The agents start with an initial amount of energy and meet randomly. If their energy is below a threshold, they fight by comparing their fitness – better agents take energy from worse ones. Otherwise, the agents reproduce and yield a new one – the genotype of the child is derived from its parents using variation operators, it also receives some energy from its parents. The system is stable as the total energy remains constant, but the number of agents may vary and adapt to the difficulty of the problem.

As in other evolutionary algorithms, agents can be split into separate populations, called islands. Such islands help preserve diversity by introducing allopatric speciation and can also be executed in parallel. Information is exchanged between islands through agent migrations.

In Krzywicki *et al.* (2015), an efficient and scalable implementation of EMAS was proposed, based on the concept of so-called arenas, depicted in Fig. 1 as Fight, Death,

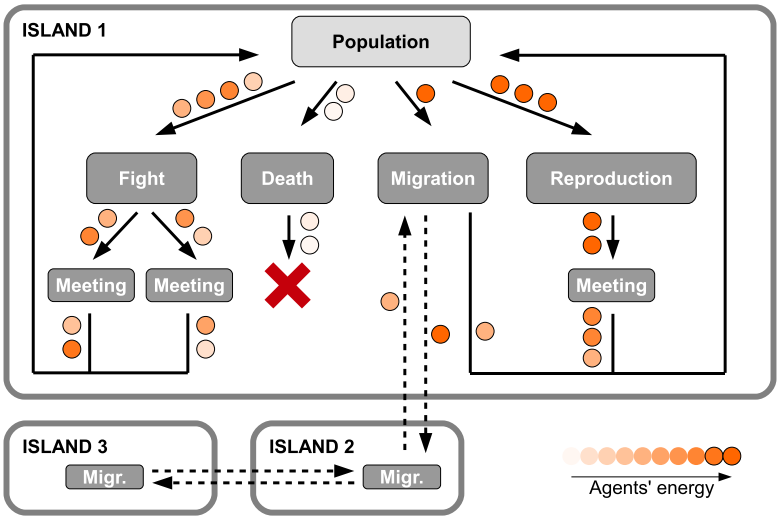

Fig. 1. Structure and behaviour of agents in the ParaPhrase implementation EMAS: interaction among individuals and arenas, as well as other islands.

Migration and Reproduction curved rectangles. The arenas (implemented as actors) are used to group agents willing to perform similar actions. Agents choose and join an arena depending on their state by exchanging the messages with those arenas to be accepted there. The parameter deciding, whether an agent is to be accepted to the arena is connected with the system idea (e.g. in the implementation of Evolutionary Multi-Agent System this is the agent's amount of energy). Arenas split incoming agents into groups of some given cardinality and trigger the realization of actual actions. Every kind of agent behaviour is represented by a separate arena. More than one arena may be present at an island in order to decentralize and speed-up the communication between the agents. Thus the way towards reaching high scalability is paved.

The actual behaviour of the particular agents interacting with the arenas shown in Fig. 1, is defined by several functions. The first function represents the agent's autonomous decision and returns an action to be performed. The action is mapped to a specific type of arena for each agent (mapping step). The other functions, representing the meeting logic, are applied in different arenas (processing step). This approach is similar to the MapReduce model and has the advantage of being very flexible, as it can be implemented both in a centralized and synchronous way or in a decentralized and asynchronous way (Krzywicki *et al.*, 2015).

Summing up, the agents employed in the architecture of the discussed EMAS implementation belong to three classes:

- Lightweight solution agents – they represent the solutions and contain a decision algorithm that leads them to perform subsequent actions. The actions cannot modify the environment directly and require the intervention of the arena agent.
- Arena agents – they accept the incoming agents and allow them to perform the actions, which include the changes of the energy level. The actions include: fighting other agent

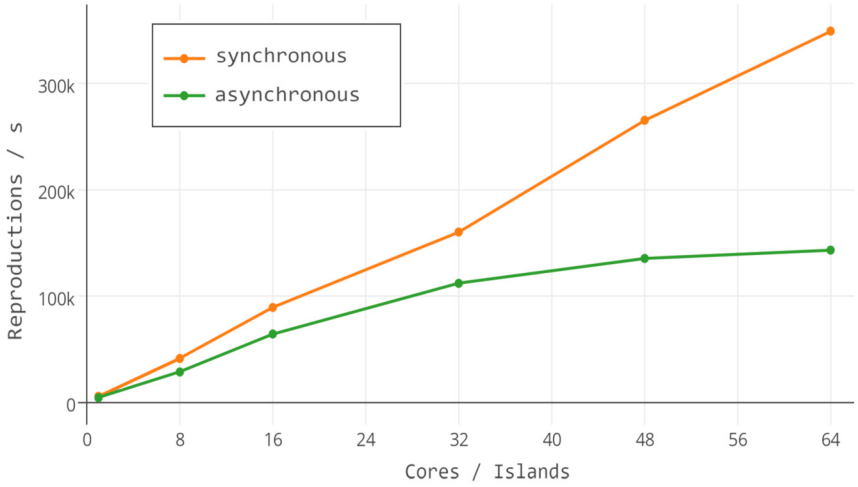

Fig. 2. Computing performance of synchronous and asynchronous implementation of the Evolutionary Multi-agent System (Turek *et al.*, 2016).

to take a portion of its energy, reproducing with other agent to create a new agent, migrating to another island, and dying, which removes the agent from the environment.

- Island agents – heavyweight agents, responsible for processing of the whole agent population, and can be treated as an environment for the individual agents (here – lightweight solution agents).

Two different implementations of EMAS have been compared in Turek *et al.* (2016): synchronous and asynchronous. The asynchronous approach adopted the most typical approach for implementing an agent system. Each solution agent and each arena agent were implemented as separated processes of the Erlang[1] virtual machine. This technology provides a preemptive scheduler, which guarantees fair access to computing resources. As a result, each agent had the same chance to perform computations, the order of agents' actions was not specified and the actions were taken in parallel on a multi-core CPU. Agent sent messages to arenas, willing to interact with other agents. The arenas sent back the results of the interactions.

The synchronous implementation reproduced the similar effect using fewer processes. The solution agents were represented by data structures that contain their state. Each agent was asked to perform its actions by executing the first of the aforementioned functions. The data structures were then grouped by the returned decisions and processed by arenas. Each arena shuffled the agents to ensure randomness and executed the proper action on the solution agents.

The optimization results were similar in both cases. However, the computing performance and scalability was significantly different (Fig. 2). The asynchronous implementation, which may seem more "the agent way", is nearly an order of magnitude less efficient than a synchronous one. More details can be found in Krzywicki *et al.* (2014).

[1] https://www.erlang.org/

Several prototypes in other technologies supported these results. Further experiments on the synchronous implementation demonstrated that it can easily be scaled in a distributed setting so that the efficiency of the algorithm increases when new nodes are added to the computation.

The concept of meeting arenas allowed to retain the expressive power of existing agent-based algorithms but led to much more efficient implementations.

3.2. *Xinuk: Discrete Multi-Agent Simulations*

The area of computer simulations is observing a substantial increase in popularity and usefulness. The social phenomena in particular benefit greatly from agent-based modelling (ABM), being able to represent the modelled beings with a degree of autonomy, expressed mainly by an ability to observe a limited fragment of an environment and make decisions in order to achieve some predetermined or dynamically changing goal. As the variety of research topics that can be modelled grows over time, so does the complexity of those models, either due to the scope of the modelled environment, a granularity of a model, or the amount of data required to describe the entities.

In order to address the increasing needs in computing power and memory, the Xinuk (Bujas *et al.*, 2019; Paciorek and Turek, 2021) framework was created, designed for the creation and execution of distributed discrete agent-based simulations. This framework utilizes the concept of signal propagation (Żabińska *et al.*, 2013) to limit the range of agent interaction with environment, reducing the need of accessing remote parts of the environment. The distribution is achieved by dividing the environment into parts assigned to different computing units (workers) and providing transparent means of communication between the workers. The process of making a decision consists of several steps, during which all of the decisions that would affect the same part of the environment (e.g. the same cell) are gathered and any conflicts are resolved so that only one of the conflicting decisions is actually applied.

Figure 3 shows the simplified view of the architecture of mechanisms that provide transparent distribution capabilities. Each of the equally sized parts of the environment is handled by a different worker (*Worker1* and *Worker2*), but agents representing the modelled entities (*A1*, *A2*, and *A3*) can freely move between the parts and are unaware of the

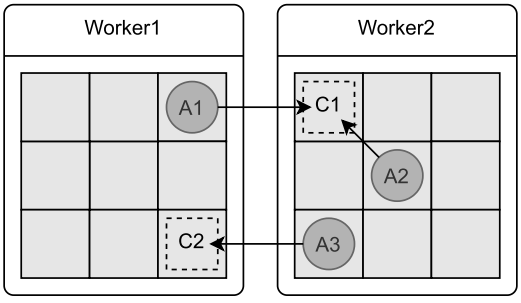

Fig. 3. Simplified architecture of Xinuk framework.

inter-worker boundaries. The action intents are handled by target cells – in this case, *C1* must resolve conflicting move intents of agents *A1* and *A2* trying to enter it, while *C2* does not detect any conflicts and is able to simply accept the move intent of agent *A3*.

The concepts underlying the implementation of Xinuk include three constructs that fit the definition of an agent:

- Each worker is given the ownership of a part of the environment and is tasked with progressing the simulation. Workers communicate with each other via messages to ensure the state of the whole environment is consistent and any change happening at the boundary is propagated to the proper worker. Additionally, each worker internally processes its part of the environment, giving the entity agents an opportunity to make their decisions and resolving them. The worker agent has its own thread.
- Each modelled entity is assigned some goal, depending on the specific model. It can observe its neighbouring environment (e.g. its own and adjacent cells) and, based on the observation and some internal state, make an independent decision. The entity agent does not control its own thread, but receives a brief autonomy from the worker agent when invoked to signal its next action intent.
- Each discrete part of the environment (e.g. each cell) collects all entity agent action intents that affect that cell. After all intents are gathered, it ensures that all intended actions that can be executed simultaneously are applied, while all intents conflicting with previously applied ones or with the current cell state are rejected. This conflict resolution agent does not control a thread and is invoked by a worker agent once in each iteration.

The distribution of the workload between the worker agents ensures massive scalability, close to linear for thousands of computing cores. The crucial results from the performance and sclability evaluation, conducted on a supercomputer, are collected in Fig. 4. At the same time, the framework is able to accommodate a variety of real-life scenarios, providing unique benefits stemming from the agent-based approach to the modelling and hiding the distribution from the model and entity agents.

3.3. *MATSim: Modelling of Network Traffic*

Studying certain phenomena in various types of networks is a big challenge, which requires detailed models and tools for their simulation. Nowadays, one of the very popular solutions in the fields of computer networks analysis and urban traffic simulations is MATSim (Horni *et al.*, 2016), developed for many years as an agent-based, open-source framework. With the use of this framework, the environment model can be imported from real-world data, like OSM[2] map and represented in the form of a graph. Each lane, represented by an edge between adjacent nodes, is modelled in the form of a queue – the model is inspired by the FASTLANE proposed by in Gawron (1998). Each lane has a set of parameters describing its features and state, such as "storage", which determines how many vehicles are able to fit into it, "flow", which tells how many are able to leave it at the

[2] https://www.openstreetmap.org/

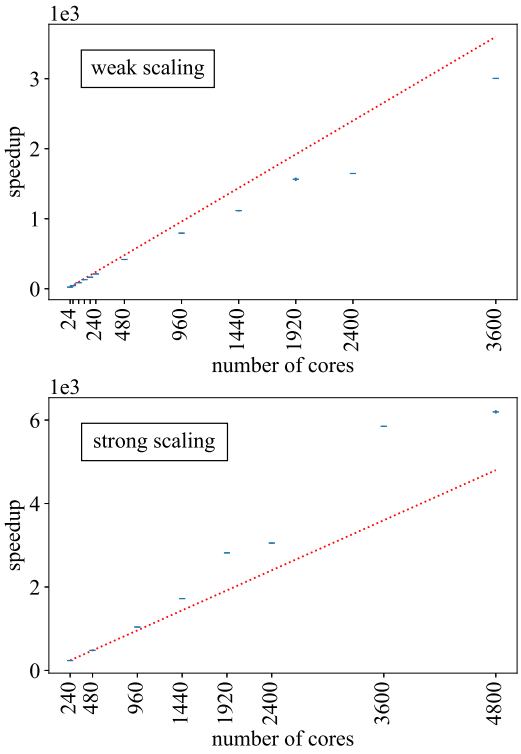

Fig. 4. Scalability of the Xinuk simulation framework (Paciorek and Turek, 2021).

moment, and "free speed", which limit maximal throughput of the lane. Figure 5 shows an exemplary multigraph of lanes and crossroads.

Each lane can be occupied by a set of agents representing cars. The cars are organized in a queue. Each of them represents a single person making decisions based on predefined plans.

To utilize the computing power of many processors and allow the system to simulate large scenarios, a few attempts have been made to introduce a distributed architecture for the simulation. The general distribution architecture of the parallel MATSim is presented in Fig. 5. It consists of a Master module and several Workers. Each worker is assigned a set of crossroads and all their incoming lanes. The workers communicate with each other by sending outgoing cars. They also send event to the Master module about all cars changing their lane. The plans of individual agents are recalculated every day (after a predefined number of simulation steps) by a centralized planning module, using a coevolutionary algorithm. The re-planning is done in a centralized manner due to the fact that it requires knowledge from the whole system.

Taking a closer look into MATSim implementation, several types of agents can be identified:

- Each person or vehicle is represented by a stateful agent, making decisions based on their current plan and state of the neighbouring environment. The agent operates within

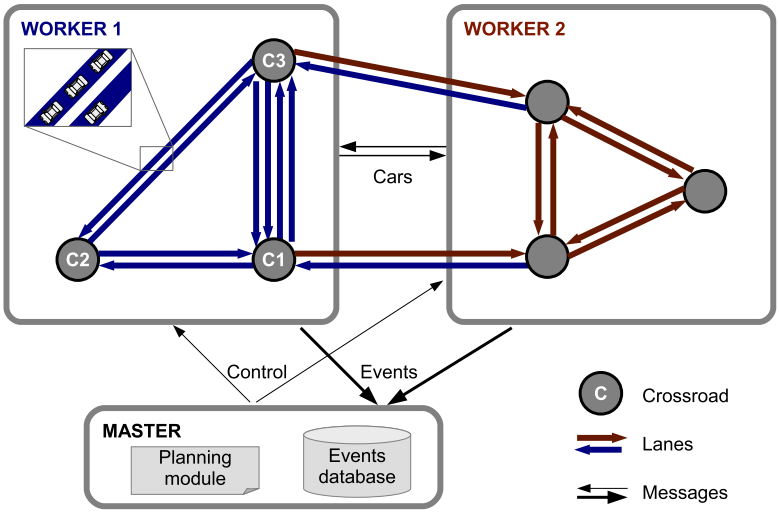

Fig. 5. Streets' model and its distribution architecture in MATSim.

graph edge that represents a single lane within the road. It can analyse its own road and roads entering and exiting the next crossing. With all that information each agent makes an independent decision. It can change its speed, change the lane or report the will to cross an intersection.

- When vehicles approach a crossroad, which is a node in a graph directed by a few edges, they are transferred to target lanes – added to the target queue. Before it happens, the intersection has to decide which agents can cross in the current step. This decision is computed using specified rules, e.g. traffic lights.
- Each worker controls its work area assigned by the graph partitioning algorithm. The worker operates within those environmental boundaries allowing vehicle entities to change their state if needed. Each worker is internally multi-threaded, using many cores to update all the lanes and intersections. When a vehicle wants to be moved between road nodes assigned to different computational nodes, the worker must communicate with its adjacent worker. The source worker sends a message containing the state of the vehicle and identifiers of the source and destination road node. To reduce the number of exchanged messages, workers are able to aggregate vehicles with the same source and destination road sections into a single message.

Selected results of the scalability experiments, which were described in Janowski (2021), are presented in Fig. 6.

The architecture of the distributed multi-agent system seems similar to the previously presented solutions. The workers manage a part of the environment and a subset of all cars. They can migrate selected cars, when needed. The cars are updated according to their individual state by the crossroads. The crossroads have to solve the conflicts, deciding on traffic priority. Processing of crossroads can be efficiently executed in parallel within a single Worker. All Workers also can work independently, exchanging limited number of messages with their neighbours.

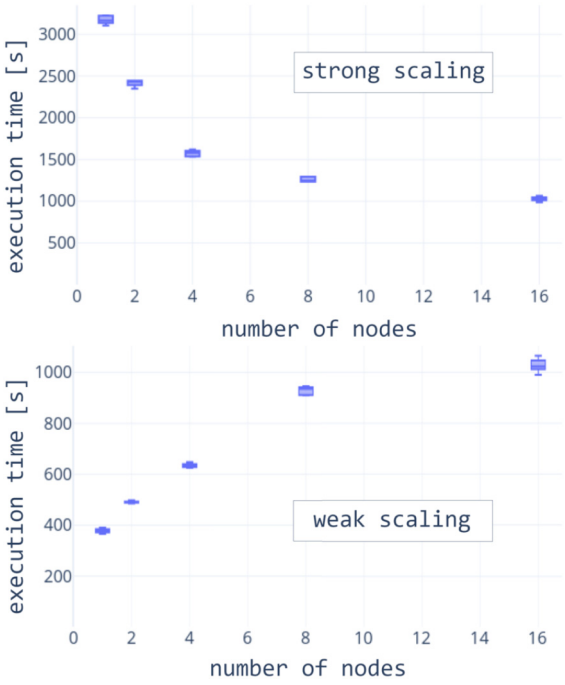

Fig. 6. MATSim scalability with centralized events processing disabled (Janowski, 2021).

Unfortunately, the parallelization of this system is profitable only up to 16 computing nodes. The presence of the master component, which requires centralized synchronization of the simulation, is definitely the reason for the scalability problems here. The results are even worse when a mechanism of centralized events processing is enabled. The mechanism requires all the events to be collected and processed by the master agent in order to perform so-called *within day re-planning*. Such mechanisms, although simple and efficient in small scenarios, always cause problems at scale.

Apart from the examples described in this section, we came to similar conclusions when building or analysing several other agent-based systems, e.g. Najdek *et al.*, 2021. The findings and observations presented further are equally valid for these and many other cases.

The analysis of the described multi-agent systems led us to the conclusion that the same mechanisms emerge as solutions to similar problems related to scalability and management of large quantities of agents. Although the context varies from application to application and the naming convention differs, the core idea behind those solutions remains largely the same, i.e. there is omnipresent need for efficient localization of the agents, scalable communication among them, creation and removal of the elements of the computing or simulation in the frames of the system structure and of course delivering the dedicated computing power (required hardware and memory) to each of the parts of the system. It is safe to assume that the design process in each case required work from the ground up, reinventing the solutions to the common problems.

4. Design Patterns for Scalable MAS

The analysis presented in the previous section inspired us to extract the common core concept from each approach applied in those systems and present them in the form of design patterns. This systematization should make it possible for future designers of systems tackling similar problems to find the proper solution quickly and easily.

Following the schema presented in Weiss (2003) we describe each design pattern using the following elements:

- *Context* – the general qualities or requirements of the system being designed,
- *Problem* – the specific challenge the pattern aims to address,
- *Forces* – the requirements imposed on the solution, often including qualities that contradict each other (which precludes the existence of trivial solution),
- *Solution* – the detailed description of the approach that can be applied to solve the *Problem* while balancing the *Forces*,
- *Variants* – optional description of more specialized applications of the design pattern,
- *Consequences* – summary of the advantages provided by the application of the design pattern, as well as any disadvantages linked to its usage,
- *Examples* – sample occurrences of the design pattern in the systems presented in the previous section.

4.1. *Colony Design Pattern*

Context. When the system has a collection of Colony Members with similar states that can often be lightweight but not exclusively. They need to cooperate and autonomously interact with each other within the system to solve a problem they face.

Problem. Implementing efficient management of a huge number of independent Colony Members within the restricted collection.

Forces.

- The Colony Members should keep agent primary characteristics such as its *autonomy* where decision making does not depend on human interaction, its *communicativeness* meaning ability to cooperate with other Colony Members and its *perception* in context of the ability to observe and react accordingly to changes in the environment.
- Individual Colony Member state should be encapsulated and may not be changed from an external source, but should be modifiable only by its own actions.
- Colony Members should operate independently of each other to allow them to be executed asynchronously. They cannot depend on each other's consecutive actions.
- High computing performance should be ensured.

Solution. The Colony pattern assumes some feedback information from interaction with the environment, which can be implemented in many ways. One approach is to share an environment between the Colony Members, where each member operates on the shared

memory space. Another implementation assumes that the Colony Member is injected with information about the current state of the environment. The Colony Member can also obtain this information by requesting changes from the observed environment.

The Colony Members can operate when they obtain some computing time of a CPU. The number of processing threads that assign control to the Colony Member for the purposes of a single decision should be strictly restricted. The Colony can use a single thread or a dedicated pool of threads for processing. It manages this thread pool and assigns computing time to each Colony Member at its discretion. The Colony Members should not be provided with dedicated threads for each member, as such a solution does not scale well, instead they should share existing resources.

A Colony Member is not able to directly interact with other Colony Members and change their state or directly affect the state of the environment. Instead, it can observe the environment with limitation to its requirements, and based on that feedback and supplied algorithm, it can change its inner state. The Colony Member is also able to report an action intent for interaction with the environment or with another Colony Member, to which the Colony reacts by transferring it to the dedicated conflict-resolving space, such as, e.g. Arena, described in the following section.

Consequences. The Colony is scalable, since from the point of view of every light Colony Member entity in the system, the size of the Colony does not change the number of operations and the work to be performed. This approach is much more efficient than thread per single Colony Member, which allows for such scalability. It also allows all Colony Members to sustain the agent characteristics. This Colony paradigm gets rid of bottlenecks, because there is no synchronization on shared resources needed, as they are not even able to directly modify shared resources. Interaction between Colony Members requires a mediator to exchange information between them. It does not allow for mutually exclusive interactions between Colony Members; if they want to interact with each other, an Arena is required.

Examples.

- In ParaPhrase EMAS each agent describes a potential solution to some optimization problem. Such an agent receives autonomy in the form of processing time by exhausting its assigned energy level. Agent decisions are made by a contained decision algorithm. Some decisions can result in interaction between the agents that can be performed only by appropriate arenas, whereas the result the agents modifies its state, thus changing the parameters of its decision algorithm.
- Xinuk represents each individual in the population, allowing them to make autonomous decisions based on observation of a limited fragment of the environment. The goal of each agent is to progress the simulation. They reside in a single discrete cell, receiving autonomy once per iteration with the ability to make an intent if they want to interact indirectly with other beings, which can result in a success or a failure.
- MATSim agents in the context of urban traffic are represented by vehicles that can make a decision based on their current plan and the state of the environment. They contain

their own state and can be invoked to update separately, while being able to interact with each other and influence the behaviour of other agents. To do that, they need to make an intent such as a lane change or turning at an intersection.

4.2. *Arena Design Pattern*

Context. As long as the activities of the agents are performed separately and do not affect the state of other agents or the environment, the task of updating the state of the system over time can be efficiently implemented. However, in the majority of situations, direct interactions between agents (like meetings, exchange of knowledge, conflict resolution, fighting) are necessary. When a multi-agent system is composed of a large number of agents and the actions of agents influence their common environment or other agents, the process of resolving state changes in the agent system becomes non-trivial. Having a common state, which may be modified by different entities, requires synchronization, which directly leads to limited scalability.

Problem. Implementing efficient interaction and conflict resolution mechanisms in action intents reported by different agents in a multi-agent system.

Forces.

- The system is composed of a large number of interacting agents.
- There are interactions between the agents and the environment that are mutually exclusive.
- There are interactions between agents that result in a state change affecting more than one of the agents.
- The interactions must be fair or reflect designed privileges.
- Efficiency of the system disallows global processing of interactions and exclusive locks on all common resources.

Solution. The action intents from all agents are collected by the Colony and the corresponding members are routed to dedicated Arenas. An Arena Agent is an algorithm for resolving conflicts related to mutually exclusive actions and interactions influencing several agents. An Arena is responsible for providing fairness or obeying priorities, which can be achieved by shuffling or sorting incoming agents. It authoritatively decides what the results of the interaction are and informs the involved agents about it.

An Arena can also be an implementation of a meeting place, a service for finding other agents, or publishing offers. Agents willing to cooperate with each other report such intent and are interconnected by the arena.

The key feature of the Arenas is their independence from each other. There is no interaction between Arenas during the evaluation of intents. All intents regarding the same element of the environment have to be processed by a single Arena, which may lead to centralized processing. However, in a properly designed system, it is typically possible to identify separate situations and elements of the environment, which may and should be managed by separate Arenas.

Consequences. The Arenas are the means for implementing complex interactions between agents. Splitting the types and resources of interactions between the Arenas has a few consequences. It organizes the dynamics of the system into separated and clearly visible pieces of code executed by dedicated agents. More importantly, it enables scalability, as the Arenas can compute their decisions independently – all the Arenas can be processed in parallel, without blocking dependencies.

Achieving this desired architectural feature requires the proper design of Arenas. The Arenas should represent the independence of actions taking place in the agent system. All mutually exclusive action intents must be resolved in a single Arena. For example, if there is a single fork in the system and many agents need it, there has to be one Arena, which decides who gets it. Independent resources can be assigned to different Arenas – if there are two forks in the system, two arenas can be created. Typically, agents are allowed to report one intent at a time and be moved to a single Arena. As a result, a single agent cannot compete for two different forks if it can handle only one fork at a time.

Examples.

- ParaPhrase EMAS is the system that first introduced the Arena concept and used it directly to resolve all interactions between Agents, therefore we adopted the name "Arena" from it. Each evolutionary island contains a Fight Arena, a Reproduction Arena, a Migration Arena, and a Death Arena. In each iteration of the system, all agents have to report a single intent, which directs it to a proper arena. The Fight and the Reproduction Arenas randomly pair the agents and perform the interactions between them. The other two Arenas process each agent separately, reducing the set of agents. This limited set of Arenas provides a clear structure of code, however, it does not support massively parallel execution, as each Arena processes agents sequentially. The scalability of this system has been achieved by using multiple evolutionary islands.
- In the case of Xinuk simulation, the concept of Arenas is adopted to resolve conflicts between agents (representing the simulated beings) willing to move to the same location. The Arenas are more fine-grained in this system – each cell of the simulated, discrete space can receive motion intents from one or more agents. The Cell Arena has to decide what is the result of multiple intents. Depending on the simulation model, some intents may be denied (agents cannot share the same location) or an interaction between agents can be triggered (e.g. one agent eats the other). Fine-grained Arenas allow both sequential and parallel processing, which significantly supports scalability.
- A fine-grained structure of Arenas is present in the MATSim network traffic simulator, where each node of the network processes agents waiting in the incoming queues. Each Node Arena implements rules or priorities according to the model definition. In the case of urban traffic, the Arena may implement traffic light sequences to select passing agents. In the case of a computer network, the priorities of packet channels can be considered. Following the Arena pattern's assumption on independence, the nodes can be updated in parallel, allowing efficient utilization of multi-core hardware.

4.3. *Breeders Design Pattern*

Context. The system uses a large number of agent entities and requires a subdivision of them into smaller groups. The subdivision is not predefined and can be dynamically changed. Depending on the requirements of the system, the assignment of the entities to the subsets should correspond to an appropriate change in the observable environment or be unnoticeable to those entities.

Problem. Implementing mechanism of division and migration of entities between subsets.

Forces.

- The system relies on managing a large quantity of entities.
- The entities need to be divided into smaller subsets.
- The entities should not be responsible for managing their allocation to the subset, this process should be hidden from them.
- The entities must preserve their ability to make decisions and observe the environment.
- The subdivision corresponds to some topology of communication between subsets.

Solution. Appoint mediator agents (Breeders) that serve as containers for the subsets. Each of the Breeders takes responsibility for the consequences stemming from the subdivision – facilitates incoming and outgoing migrations and ensures access to the environment if it is also subject to the division. Any interaction of the entities within a single Breeder can be resolved locally, while any interaction between entities within different Breeders requires communication between them. This communication should be conducted transparently from the point of view of the entity, as it should not have to be aware of any type of communication across Breeder boundaries.

Each Breeder can communicate with other Breeders, allowing for cooperation. The communication should be limited to the local neighbourhood, ideally constant in size regardless of the number of Breeders, to avoid the significant increase in required interactions as the number of Breeders increases. Failure to meet this requirement might significantly hamper the performance of the system, as each Breeder would need to consult its decisions with all other Breeders. Furthermore, this neighbourhood should directly mirror the intended topology of the communication between the subsets of entities.

Variants. This design pattern combines two important, but noticeably different variants.

The first variant – Logical Breeders – occurs, when the subdivision is important from the system logic point of view. The Breeders, in addition to monitoring and aiding in migrations of the entities, ensures proper isolation from the entities within other Breeders. In this case, the usual purpose of the communication between Breeders is to determine the proper destination of the migrating entity, according to the logical constraints of the system. In this case, the Breeders should be limited in communication capability to the logical neighbourhood between subsets of entities, emerging from the requirements of the system that resulted in the need for division.

The second variant – Transparent Distribution Breeders – ensures better scalability of the system. In this case, the subdivision is not a part of the system logic and should remain completely transparent for the entities. If the environment represents some dimensional, spatial area, it will usually also be divided. This gives the Breeder the responsibility for providing the subset of entities with the proper section of the environment, as well as detecting an intent that should result in the migration of an entity. The communication between Breeders should be limited to the nearest neighbourhood determined by the geometry of the division of environment, i.e. corresponding to the boundaries that require remote communication to ensure continuity of the environment.

Consequences. Application of this design pattern provides the tools necessary for coordination of the many subsets of a single Colony. In addition, it intrinsically facilitates the distribution of the system, as the existence of Breeders allows them to take over remote communication and aggregate messages that could otherwise be sent individually by each of the entities. Constant amount of connections between Breeders ensures that increasing the number of Breeders will not increase the cost of communication for each of them. This quality is required to ensure the massive scalability of the solution (Engelmann and Geist, 2005). As a result, the existence of any boundaries between Breeders can be hidden from the entities (in the case of Transparent Distribution Breeders) or can be properly taken into consideration by entities (in the case of Logical Breeders).

Examples.

- ParaPhrase EMAS utilizes the concept of islands, which contain some limited population of problem solutions represented by agents, which fits perfectly into the definition of Logical Breeders. Any action that can be performed is limited to a single island, except for the migration intent. The need for migration is explicitly signaled by the solution agents to the dedicated migration Arena, which in turn defers the resolution of the migration process to the island.
- In Xinuk, the idea of workers is an example of Transparent Distribution Breeders. Each worker handles some part of environment and the agents present in this part of environment. The migration between workers is hidden from the agents, as well as the fact that given observation of environment was performed on a boundary between workers.
- MATSim utilizes the concept of Transparent Distribution Breeders, as the environment, similarly as in Xinuk, is divided appropriately between all workers. The agent is not aware of affiliation with Breeder responsible for it.

5. Cooperation of the Design Patterns

Each of the presented design patterns can be used separately or together with other patterns defined for different purposes. Each of them can improve efficiency and scalability on modern multi-core and multi-node hardware.

The Colony pattern can be used separately for improving the performance of agent-based systems with a large number of lightweight agents, provided that their interactions

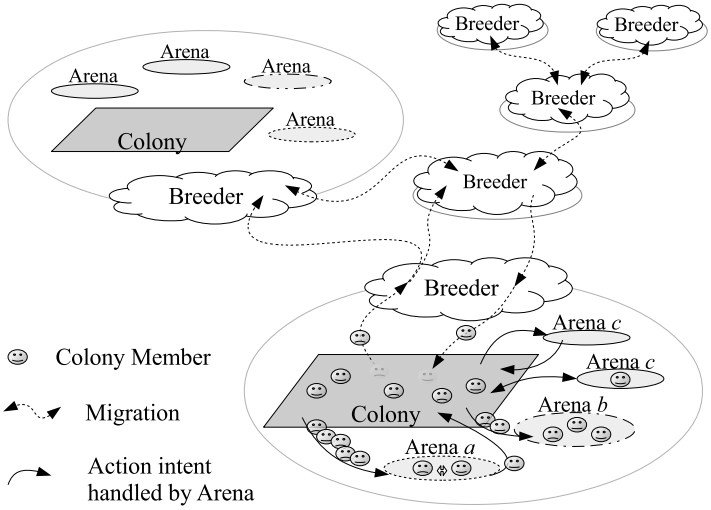

Fig. 7. Scalable MAS architecture based on the three design patterns: Colony, Arena and Breeders.

are not mutually exclusive. One could imagine a simulation of a forest, where each tree-agent updates its own state (grows, drops branches) and influences the environment in a non-exclusive manner (generates seeds, drops shadow). Such agents can be efficiently updated, either sequentially, or in parallel, by several threads simultaneously.

The problem of managing interactions, solved by the Arena pattern, is conceptually similar to solutions often referred to as Mediator (Juziuk *et al.*, 2014). However, the focus of the Arena pattern goes beyond handling interaction. Properly identified mutually exclusive and independent actions in an agent-system, enclosed in separated, fine-grained Arenas allow efficient utilization of multi-core hardware by performing selected interactions simultaneously.

Running a consistent multi-agent system on many computing nodes is the desired possibility in many areas. Building a proper topology of Breeders pattern, each handling migrations from and to a well-defined part of agents' population, turned out to be a successful and scalable approach. By controlling the number of Breeders' neighbours and limiting their responsibilities to migrating agents, linear scalability up to hundreds of HPC nodes can be achieved.

However, the three patters are designed to cooperate together within a single system. All three of them are focused on supporting scalability of different aspects of complex multi-agent systems. Together they formulate a framework for building systems, which can represent various agent-based models and execute efficiently on concurrent and distributed hardware, including modern HPC environments. The abstract architecture of the cooperation between the three patterns in such a system is presented in Fig. 7.

The environment of agents and the population of agents are split between Colonies. Each Colony has a set of necessary Arenas to process complex interactions between agents. Each Colony is observed (or managed) by a Breeder, which provides means for exchanging agents between Colonies. The Breeder can create a "virtual common environ-

ment" for Colony Members (Transparent Distribution Breeders), or explicitly model the existence of separated sub-populations, whichever is required by the agents system model.

There are a few ways of implementing the control flow between the patterns. In the simple case, the processing may be performed by a single thread, which iteratively executes Colony Members update, Arenas update and migrations. However, the patterns allow far more sophisticated and efficient implementations. Update of Colony members can be done in parallel by several threads; similarly, Arenas can compute their decisions in parallel. What is more, the patterns allow the non-iterative execution of these two major steps. The update of Colony Members and Arenas can be executed simultaneously and continuously, simulating mode natural time flow in the system. To achieve this, two separate pools of threads process Colony and Arenas, while Arenas wait on a sufficient number of agents to process interactions. The Breeder should typically have its own, single thread, which serves for sending and receiving messages from other Breeders.

All these means serve a common purpose: to achieve efficiency and scalability while preserving crucial features of the agent-based approach. The efficiency on a single computing node requires constant saturation of all cores with limited context switching. This has been achieved by providing many fine-grained, independent tasks, which can be executed by a pool of threads. The scalability beyond a single computing node requires asynchronous communication in topology, with constant communication cost per member. The proposed design patterns and the architecture composed of them provide these features while preserving the possibility of modelling individual agents' autonomy.

A interesting feature of the considered architecture is the presence and coexistence of completely different classes of software agents inside it. By completely different we do not mean different agents coexisting in a common environment. On the contrary, it would be more suitable to identify several clearly separated multi-agent systems in a single solution. While trying to identify abstractions from these observations, we came to an analogy from the surrounding world. If a typical system with coexisting agents can be seen as a population of people and animals sharing the same village, the relations between the agent-based beings in the proposed architecture recall a town of people, each having a population of microbes inside. Being mutually unaware of each other's existence and having completely different awareness, environment, means of communication, and aims, they create an efficiently working system. At the same time, both a *Homo sapiens* and a *Bifidobacterium bifidum* perfectly fit the commonly used definition of an agent, stressing autonomy as one of the most important agent's features. In the proposed approach, the Breeders, Colonies and Arenas can be considered agents whose motivation and purpose is to create an environment tor the numerous, tiny agents "living inside" them.

6. Conclusions and Further Work

Being aware of different agent-based design patterns, our aim was to propose three new ones, not only connected to actual application (solving certain substantial problem, like making possible realization of certain agent-based simulation), but to support the efforts

in efficient leveraging of the hardware infrastructure. This problems become a significant challenge when number of agents grows, their interactions become numerous and influence common environment.

Thus, based on an analysis of several different agent-based computing and simulation systems, the *Colony*, *Arena* and *Breeders* design patterns were identified. The patterns are not invented by analysing the needs. On the contrary, they emerged in systems which were actually implemented, having in mind the scalability on HPC infrastructures. Their definitions, presented in this work, have been created by abstracting crucial features of the scalable solutions. An effort has been made to describe them in a form used in other State-of-the-Art papers, and pave the way for further extension of this set of agent-oriented design patterns aimed at achieving concurrency and scalability of the constructed multi-agent systems.

It is to note that the proposed patterns can be used separately, however, they provide the most advantages when used together. The proposed set of patterns can be further extended (keeping in mind the scalability and efficiency of the constructed agent-based systems), and indeed further research will focus on e.g. employing and standardizing the usage of e.g. computing accelerators (e.g. GPGPU or FPGA), however, it would be great if other researchers, dealing with the construction of similar systems, could add other patterns to this proposed initial set.

Funding

The research presented in this paper was funded by the National Science Centre, Poland, under the grant No. 2019/35/O/ST6/01806 (MN). The research presented in this paper was partially supported by the funds assigned by the Polish Ministry of Education and Science to AGH University of Krakow (WT, MP). We gratefully acknowledge the funding support by programme "Excellence initiative–research university" for the AGH University of Krakow as well as the ARTIQ project: UMO-2021/01/2/ST6/00004 and ARTIQ/0004/2021 (AB).

References

Aridor, Y., Lange, D. (1998). Agent design patterns: elements of agent application design. In: *Proceedings of the International Conference on Autonomous Agents*, pp. 108–115. https://doi.org/10.1145/280765.280784.

Asghar, M.Z., Alam, K.A., Javed, S. (2019). Software design patterns recommendation: a systematic literature review. In: *2019 International Conference on Frontiers of Information Technology (FIT)*, Islamabad, Pakistan, 2019, pp. 167–1675. https://doi.org/10.1109/FIT47737.2019.00040.

Barrera, R.S., Gómez, G.R., López-López, A. (2011). Design patterns in multi-agent system simulation. In: *CONIELECOMP 2011, 21st International Conference on Electrical, Communications, and Computers*, San Andres Cholula, Mexico, 28 February–2 March 2011, pp. 60–65. https://doi.org/10.1109/CONIELECOMP.2011.5749340.

Boronea, S., Leon, F., Zaharia, M.H., Atanasiu, G.M. (2009). Design patterns for multi-agent simulations. *Management and Marketing*, 4(4), 15–26.

Bujas, J., Dworak, D., Turek, W., Byrski, A. (2019). High-performance computing framework with desynchronized information propagation for large-scale simulations. *Journal of Computational Science*, 32, 70–86.

Byrski, A., Dreżewski, R., Siwik, L., Kisiel-Dorohinicki, M. (2015). Evolutionary multi-agent systems. *The Knowledge Engineering Review*, 30(2), 171–186. https://doi.org/10.1017/S0269888914000289.

Cetnarowicz, K., Kisiel-Dorohinicki, M., Nawarecki, E. (1996). The application of evolution process in multi-agent world to the prediction system. In: *Proceedings of the Second International Conference on Multiagent Systems*. AAAI, pp. 26–32.

Cruz Torres, M.H., Van Beers, T., Holvoet, T. (2011). (No) more design patterns for multi-agent systems. In: *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11. SPLASH'11 Workshops*. Association for Computing Machinery, New York, NY, USA, pp. 213–220. 9781450311830. https://doi.org/10.1145/2095050.2095083.

Engelmann, C., Geist, A. (2005). Super-scalable algorithms for computing on 100,000 processors. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (Eds.), *Computational Science – ICCS 2005, ICCS*, *Lecture Notes in Computer Science*, Vol. 3514. Springer, Berlin, Heidelberg, pp. 313–321. https://doi.org/10.1007/11428831_39.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA. 0201633612.

Gawron, C. (1998). *Simulation-Based Traffic Assignment. Computing User Equilibria in Large Street Networks*. PhD thesis, Universität zu Köln, Germany.

Hofstede, G.J., Chappin, E. (2021). Archetypical patterns in agent-based models. In: Ahrweiler, P., Neumann, M. (Eds.), *Advances in Social Simulation, ESSA 2019, Springer Proceedings in Complexity*. Springer, Cham, pp. 313–332. https://doi.org/10.1007/978-3-030-61503-1_31.

Horni, A., Nagel, K., Axhausen, K.W. (2016). *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, London.

Janowski, M. (2021). *Scalable Urban Traffic Simulation Algorithm for MATSim System*. Master's thesis, AGH University of Science and Technology in Krakow, Poland.

Juziuk, J., Weyns, D., Holvoet, T. (2014). Design patterns for multi-agent systems: a systematic literature review. In: Shehory, O., Sturm, A. (Eds.), *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 79–99. 978-3-642-54432-3. https://doi.org/10.1007/978-3-642-54432-3_5.

Klügl, F., Karlsson, L. (2009). Towards pattern-oriented design of agent-based simulation models. In: Braubach, L., van der Hoek, W., Petta, P., Pokahr, A. (Eds.), *Multiagent System Technologies, MATES 2009*, *Lecture Notes in Computer Science*, Vol. 5774. Springer, Berlin, Heidelberg, pp. 41–53. https://doi.org/10.1007/978-3-642-04143-3_5.

Krzywicki, D., Stypka, J., Anielski, P., Faber, Ł., Turek, W., Byrski, A., Kisiel-Dorohinicki, M. (2014). Generation-free agent-based evolutionary computing. In: Abramson, D., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M.A. (Eds.), *Proceedings of the International Conference on Computational Science, ICCS 2014*, Cairns, Queensland, Australia, 10–12 June, 2014. *Procedia Computer Science*, Vol. 29. Elsevier, pp. 1068–1077. https://doi.org/10.1016/j.procs.2014.05.096.

Krzywicki, D., Turek, W., Byrski, A., Kisiel-Dorohinicki, M. (2015). Massively concurrent agent-based evolutionary computing. *Journal of Computational Science*, 11, 153–162.

Lämmel, R., Visser, J. (2002). Design patterns for functional strategic programming. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, RULE'02*. Association for Computing Machinery, New York, NY, USA, pp. 1–14. 1581136064. https://doi.org/10.1145/570186.570187.

Lind, J. (2003). Patterns in agent-oriented software engineering. In: Giunchiglia, F., Odell, J., Weiß, G. (Eds.), *Agent-Oriented Software Engineering III*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 47–58. 978-3-540-36540-2.

Mathieu, P., Morvan, G., Picault, S. (2018). Multi-level agent-based simulations: four design patterns. *Simulation Modelling Practice and Theory*, 83, 51–64. https://doi.org/10.1016/j.simpat.2017.12.015.

Mohamad, R., Deris, S., Ammar, H.H. (2007). Pattern-oriented design for multi-agent system: a conceptual model. *Journal of Object Oriented Technology*, 6(4), 55–75.

Najdek, M., Xie, H., Turek, W. (2021). Scaling simulation of continuous urban traffic model for high performance computing system. In: Paszynski, M., Kranzlmüller, D., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M.A. (Eds.), *Computational Science – ICCS 2021*. Springer International Publishing, Cham, pp. 256–263. 978-3-030-77961-0.

North, M.J., Macal, C.M. (2011). Product design patterns for agent-based modeling. In: Jain, S., Creasey, R.R., Himmelspach, J., White, K.P., Fu, M.C. (Eds.), *Winter Simulation Conference 2011, WSC'11*, Phoenix, AZ, USA, December 11–14, 2011. IEEE, pp. 3087–3098. https://doi.org/10.1109/WSC.2011.6148008.

Oluyomi, A., Karunasekera, S., Sterling, L. (2007). A comprehensive view of agent-oriented patterns. *Autonomous Agents and Multi-Agent Systems*, 15(3), 337–377.

Paciorek, M., Turek, W. (2021). Agent-based modeling of social phenomena for high performance distributed simulations. In: Paszynski, M., Kranzlmüller, D., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M.A. (Eds.), *Computational Science – ICCS 2021, ICCS 2021*, *Lecture Notes in Computer Science*, Vol. 12743. Springer, Cham, pp. 412–425. https://doi.org/10.1007/978-3-030-77964-1_32.

Sauvage, S. (2004). Design patterns for multiagent systems design. In: Monroy, R., Arroyo-Figueroa, G., Sucar, L.E., Sossa, H. (Eds.), *MICAI 2004: Advances in Artificial Intelligence*, *Lecture Notes in Computer Science*, Vol. 2972. Springer, Berlin, Heidelberg, pp. 352–361. https://doi.org/10.1007/978-3-540-24694-7_36.

Sicard, V., Andraud, M., Picault, S. (2021). Organization as a multi-level design pattern for agent-based simulation of complex systems. In: Rocha, A.P., Steels, L., van den Herik, H.J. (Eds.), *Proceedings of the 13th International Conference on Agents and Artificial Intelligence – Volume 1: ICAART*. SciTePress, pp. 232–241. https://doi.org/10.5220/0010223202320241.

Turek, W., Cetnarowicz, K., Zaborowski, W. (2011). Software agent systems for improving performance of multi-robot groups. *Fundamenta Informaticae*, 112(1), 103–117.

Turek, W., Stypka, J., Krzywicki, D., Anielski, P., Pietak, K., Byrski, A., Kisiel-Dorohinicki, M. (2016). Highly scalable erlang framework for agent-based metaheuristic computing. *Journal of Computational Science*, 17, 234–248.

Vaira, Ž., Čaplinskas, A. (2011). Software engineering paradigm independent design problems, GoF 23 design patterns, and aspect design. *Informatica*, 22(2), 289–317. https://doi.org/10.15388/Informatica.2011.328.

Weiss, M. (2003). Patterns for motivating an agent-based approach. In: Jeusfeld, M.A., Pastor, ó. (Eds.), *Conceptual Modeling for Novel Application Domains. ER 2003*, *Lecture Notes in Computer Science*, Vol. 2814. Springer, Berlin, Heidelberg, pp. 229–240. https://doi.org/10.1007/978-3-540-39597-3_22.

Żabińska, M., Sośnicki, T., Turek, W., Cetnarowicz, K. (2013). Robot task allocation using signal propagation model. *Procedia Computer Science*, 18, 1505–1514.

M. Najdek is a PhD student at the AGH University of Science and Technology in Krakow. His work focuses on highly efficient and scalable urban traffic simulations. During his work on continuous models of urban traffic, he explores and expands the methods of effective scaling of simulation computing.

M. Paciorek obtained his PhD in 2022 from the AGH University of Science and Technology in Krakow, Poland. His background in designing data management systems for HPC formulated a valuable basis for research on super-scalable spatial simulations.

W. Turek obtained his PhD in 2010 and DSc in 2019 from the AGH University of Science and Technology in Krakow, Poland. He works as an associate professor at the Department of Computer Science of the AGH-UST. His research topics cover a wide range of possible applications of multi-agent system, from management of robots to distributed evolutionary computing. His recent research focuses on super-scalable spatial simulations which can efficiently utilize HPC-grade hardware.

A. Byrski obtained his PhD in 2007, DSc in 2013 from the AGH University of Science and Technology in Krakow, Poland. In 2020, he obtained the title of professor from the President of Poland. He works as a full professor at the Department of Computer Science of the AGH-UST. His research topics cover metaheuristics, agent-based computing and simulation systems, and parallel and distributed computing using HPC.