An Agent-Based Framework for Complex Networks

Alexander Wendt[1(✉)], Maximilian Götzinger[1,2], and Thilo Sauter[1,3]

[1] TU Wien, Vienna, Austria
{alexander.wendt, maximilian.goetzinger, thilo.sauter}
@tuwien.ac.at
[2] University of Turku, Turku, Finland
[3] Danube University Krems, Wiener Neustadt, Austria

Abstract. A large number of research and industrial projects could benefit from a module-based development. However, these modules and the communication between them may vary from project to project. Therefore, a general middleware instead of several specialized middlewares for each domain is desired. This paper presents the ACONA Framework (Agent-based Complex Network Architecture). It is an agent-based middleware with a lightweight and flexible infrastructure. Also, it offers the possibility of evolutionary programming. Its performance is demonstrated in three applications: (i) A cognitive architecture with around 40 interconnected modules; (ii) a stock market simulator with elements of evolutionary programming; and (iii) an industry 4.0 application of a conveyor belt.

Keywords: Multi-agent system · ACONA · Middleware · Communication · Industry 4.0 · Interoperability · Distributed application platform · Internet of Things · Self-adaption · MQTT

1 Introduction

In a heterogeneous system, each system module might be developed in a particular language. The modules might be running in a distributed fashion. Modules have individual tasks like being device drivers or controllers for applications. Many research and industrial projects show a demand for module-based development.

In many cases, it is enough to use some message bus, which provides essential means of communication. The developer then implements the custom communication methods tailored for each module. In the case of a typical Smart Grid [1] and IoT [2] applications, a message bus for basic communication between sensors and actuators is often enough. Optionally, a higher instance provides the application with directives on how to control the system.

However, in projects that implement multi-agent systems, simulations [7] or cognitive architectures [3], a bare message bus as a middleware requires a very high effort of implementing the infrastructure, as they are too simple for the target application. A cognitive architecture consists of up to 30 modules, which run in cycles and interact on-demand. In this type of systems, common programming patterns are repeatedly used. It demands a flexible agent system that offers basic functionality for designing agent functions. It would allow a fast setup of an industrial network.

Published by Springer Nature Switzerland AG 2019
J. MacIntyre et al. (Eds.): AIAI 2019, IFIP AICT 559, pp. 559–570, 2019.
https://doi.org/10.1007/978-3-030-19823-7_47

This paper proposes a solution to these demands, namely, the *Agent-based COmplex Network Architecture* (ACONA) Framework. It will be shown how an agent-based middleware is created as a layer in the communication protocol MQTT. Typical for a middleware, the developer shall not take care of communication between agents. Instead, the framework provides the necessary functionality to help to implement a self-adapting complex system. Three applications will be discussed to demonstrate the flexibility and performance of the framework.

2 Related Work

In a smart grid setup, smart meters and other sensors measure the grid's state. Based on these measurements smart breakers connect or disconnect devices if overloads are detected. A SCADA system assesses and displays an overload condition, e.g., through a traffic light system. In these applications, the middleware OpenMuc [1] can be used. It relies on a component-based programming framework OSGi[1] in Java. Channels available to all components handle the data access, and the core automatically synchronizes data between modules. However, OSGi comes with a crucial drawback for agent systems as it only allows one instance of a message bus to run on a Java VM. It makes it hard to implement parallel working agents. Other middleware solutions in this area are Siemens Gridlink [4] and TU Wien Demo Facility [5], which both provide a basic low-level infrastructure based on either VertX[2] and Hazelcast[3] or Google Protobuf[4]. These frameworks offer enough functionality to manage basic communication between modules. However, they are not suited for creating a hierarchical agent system, i.e., agents with sub-functions.

A framework like the LIDA framework [6] could be used to create a complex agent system like a cognitive architecture. Unfortunately, it uses the observer pattern between modules, which is fast but also implies a strong coupling between modules. MASON [7] is a Java-based simulation tool that offers an extensive user interface and a world simulator, where actors can interact. Agents are stepped through serially in each simulation cycle. Compared to MASON, ACONA shall provide agents, whose functionality is loosely coupled, i.e., the agents shall not be seen as a single entity, but as a composition of several individual functions.

In the first version, ACONA extended the widely spread multi-agent system Java Jade [8, 9]. Jade provides containers equivalent to a message bus, where agents run. Jade agents are threads, which get their functionality from scheduled behaviors. It has the advantage that it is very customizable. It enables negotiation between agents by using the FIPA protocol. On the other side, the developer has to implement all communication from scratch. It turned out that communication methods were hard to separate from behavior classes. The purpose was to hide the communication layer.

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

[2] https://vertx.io.

[3] https://hazelcast.com.

[4] https://developers.google.com/protocol-buffers.

Finally, Jade is designed to work as a state machine in a single thread per agent, which does not allow blocking calls, making a request-response pattern hard to implement with a method call. In the end, this was not an efficient solution. While Jade proposes a system that is supposed to have many agents with simple functionality, ACONA adds much functionality within a few agents.

ZeroMQ[5] would be an attractive alternative as it offers a broker-less architecture, which fits well into an agent concept. However, each client needs an own port that has to be addressed. The system does not scale well for replicating agents.

3 ACONA Framework Model

The basic idea is to define an agent, which modifies its internal state or the environment through functions. An agent encapsulates its functions from the environment. Internally, functions share data through a data storage. The communication between functions and other agents applies an onion model, which separates the communication infrastructure from the function logic. To achieve a general agent, which can replicate, all functions can be generated entirely from a textual configuration. Figure 1 left part shows the components of an agent. Each component will be described in the following chapter.

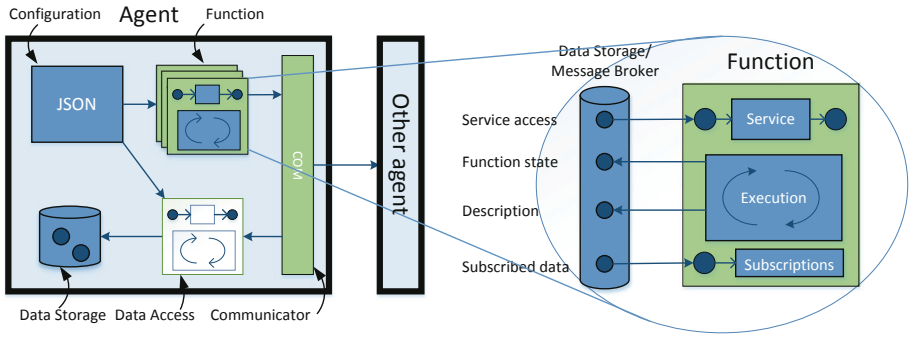

Fig. 1. Left: agent with its components; right: function features with assigned data points in a data storage

3.1 Functions

Each agent consists of a set of *functions*; see Fig. 1, right part. Anytime, a function can be registered in an agent instance to add functionality. Each function offers the following features, which implements some common development patterns:

- Services, which are accessible through other functions (request-response pattern)
- A thread, which runs code on demand or scheduled (parallel execution)
- Listeners, which are triggered by subscribed data points (publish-subscribe pattern)

[5] http://zeromq.org.

Services are accessed through a subscribed data point on a message bus, e.g., <agent1>/function1/service1. A request with parameters in JSON format that is sent to that data point. The service returns a response to a predefined callback data point. Functions offer customized services as well as some default services. Such a service is the "command" service that receives commands to start, stop or pause the function thread. The caller function blocks until it receives a response. The core functionality of a function is the running thread. It consists of three segments: pre-processing for reading data, main execution for processing and post-processing for writing data to destination addresses. A typical case is to combine services with running threads, e.g., in a controller. There, the service blocks the caller by providing a delayed response, while the thread runs its functions independently.

3.2 Communication and Data Structures

For the communication between functions, an encapsulated *communicator* is used. In contrast to the previous implementation of the framework in Java Jade, each function gets its communicator to prevent unintended locks. To make the ACONA framework natively compatible with IoT devices, we used the lightweight MQTT[6] protocol. It serializes messages bytes, uses low bandwidth and can cope with large messages. Because each function needs a communication client and the addressing of clients shall be scalable, we decided to use the Mosquitto[7] broker. Thus, it is single threaded and performance could be an issue.

MQTT only implements the publish-subscribe pattern. A communicator client publishes data to an address in the broker. Other clients subscribe to them and are notified by a callback function. However, many use cases need a request-response pattern in their communication functionalities. It is solved by using a callback address for the response.

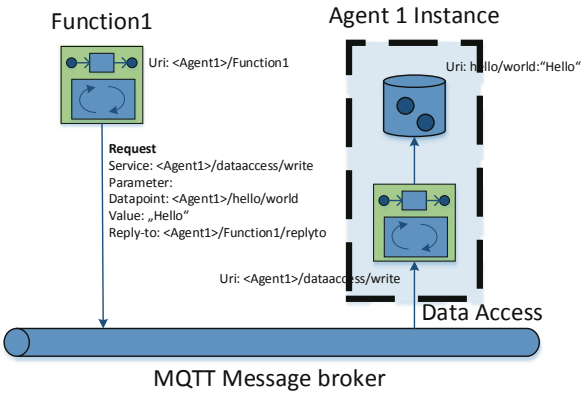

Fig. 2. A function sends a write request to the data storage

[6] http://mqtt.org.

[7] https://mosquitto.org.

It turns out that the only needed communicator functionality to develop complex systems is to implement a method that executes a service in another function.

MQTT topics define the addresses of the functions and their services. As each function belongs to an agent, the first part of the address is the agent name, followed by the function and service name. Messages use the JSON syntax to unify and simplify overhead and allow serialized objects to be passed between functions.

The desired functionality was to have a persistent database to share data between functions. MQTT does not support polling topics without subscribing them. To solve this problem, a key-value database was added to each agent instance. As all functions contain MQTT clients, the agent is defined by a common database. It uses the same addressing syntax as used in MQTT. The agent instance implements an access function for reading, writing, and subscribing values. The message broker handles subscriptions by mirroring those values. Figure 2 presents an example of how a function <function1> sends a request to its write service. The writing service writes the value <Hello> and returns a response with an acknowledgment to the caller.

3.3 Configuration

To provide usage in self-adapting industrial networks or self-healing, the system is completely configurable. A significant feature of ACONA is that all functions that define a particular agent can be composed like Lego blocks in a text-based configuration. This is possible as each function is connected through MQTT and completely decoupled from all other agent functions. Java functions are instantiated per reflections from the class and the function name. Other properties are also passed here. The composition of functions makes it easy to perform tests by replacing real functions with mockups. The message broker links functions as they share data, which makes the setup of agent networks for prototyping easy.

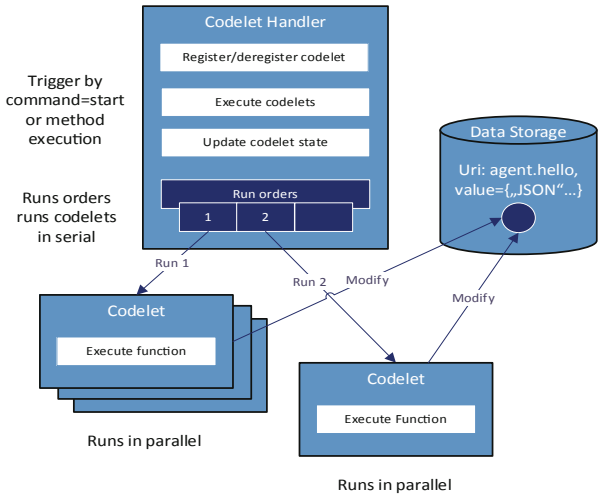

Fig. 3. Codelet handler and codelets

To enable self-adaptation of an agent system, they are equipped with a replication function. We note that an agent does not make a binary copy of itself, but it instantiates a modifiable configuration of itself or another agent. In the latter case, that would be the functionality of a virus, as it replicates a foreign agent.

3.4 Complex Structures: Controller

It is possible to develop complex functionalities like a controller. In this context, a controller is a scheduler, called a *codelet handler*. *Codelets* are independent functions that provide only an execution service and register themselves in the codelet handler. Figure 3 shows such a codelet handler with registered codelets. It provides services to register/unregister and to execute the codelets. Codelets can be executed in parallel or sequence by setting an *execution order*. All codelets with the same execution order are pooled and run in parallel, and each of these pools is executed sequentially. To instantiate codelets, they are assigned to a codelet handler function in the configuration, where also the execution order is set.

4 Application and Results

To demonstrate the flexibility and potential of the ACONA framework, it was implemented in three applications in different domains. In each application, the system complexity is presented together with a performance metric.

4.1 Cognitive Architecture in Building Automation

Enabling the ACONA framework for implementing applications of cognitive architectures in industrial applications was one primary goal of our work. A system based on cognitive architecture reads sensor inputs and provides options for various actions. These options are based on the inferred beliefs of the environment and current goals. After the system has evaluated by considering cost and gain of all possible options, it executes one of them. Instead of hard-coded functions, most reasoning is done through learning and knowledge retrieval, which makes cognitive architectures general-purpose software [3].

ACONA was used as infrastructure and middleware for the modules in the cognitive architecture of the project KORE [3]. The cognitive architecture, inspired by the cognitive model SiMA [10], was designed as a general-purpose software in the area of building automation. In a simulation of a building, each room was equipped with temperature, CO2, and occupancy detectors as well as ventilation and heating controllers. The setup was stored in an ontology. The research problem was to use the ontology to generate a set of control rules for the building automatically. The objective was to keep the comfort level while reducing energy consumption. The cognitive architecture was implemented as a controller for the generation of rules. It had the task to get optimization requests from a REST interface, select which algorithms to run for the generation and parameterization of the control strategies as well as to evaluate, how well a specific control strategy performed in the simulator. The results showed that a cognitive architecture could be applied, but it lacks efficiency as the functionality could be programmed in a straight forward manner.

Figure 4 demonstrates the complexity of such a system as it depicts the design of the cognitive architecture. A single ACONA agent implemented a top-codelet handler to run the *cognitive process*, starting from process step B to I. For each process step, a sub-codelet handler activates several parallel codelets that execute tasks such as generating an option for actions based on beliefs or making evaluating an option based on external user requests. In total, eight codelet handlers were used hierarchically, 37 codelets were running. Additionally, 20 other agent support functions were implemented. The native data storage of the agent was sufficient to use as different short-term memories. The implementation demonstrates how the ACONA framework manages to be used as infrastructure in a comprehensive cognitive architecture.

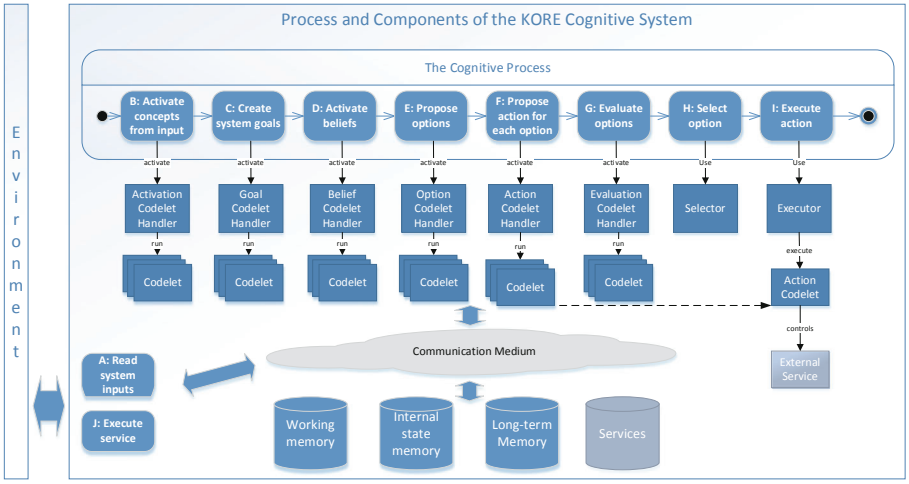

Fig. 4. The cognitive architecture of the project KORE based on the ACONA framework

4.2 Self-evolving Agents in a Multi-agent System

To explore the potential of the framework regarding self-replication, evolution, and simulation capabilities, a stock market trading game was implemented. In this game, every agent is a trader, which earns money by buying and selling a stock. Real market values represent the stock. For the test, ∼5000 closing prices of the OMXS30 index were used. For this simulation, the price patterns are relevant and therefore, data is looped to provide more extended periods than 5000 samples.

Each trader uses two exponential moving averages (EMA) as buy and sell signals, i.e., two parameters control its trading behavior. If the shorter EMA > longer EMA, the signal is to buy, otherwise to sell. The configuration defined that if a trader increases his deposit value by 30%, the agent replicates by splitting its depot into a new agent. If the deposit value decreases with 90%, the agent dies. Each of the two trading parameters "mutates" with 30% probability, 51% mutation probability for any change. Agents with equal trading parameters define a species.

An example is a species <L200S50>, where "L" is the long EMA 200 periods and "S" is the short EMA of 50. To measure the success of a species, the number of individuals per species is counted. Species that frequently replicate produce the newest species and those parameters with modifications are inherited. By producing more agents for "fitting" parameters, the evolutionary system explores different local maxima.

Figure 5 shows the setup of the stock market simulator with the ACONA framework. Step 1 to 10 describes the simulation process. First, a codelet handler executes the price generator. Then, it triggers the traders on execution order 2. They access the broker on demand for trading. Finally, the evaluation function in the broker agent is executed to update the statistics.

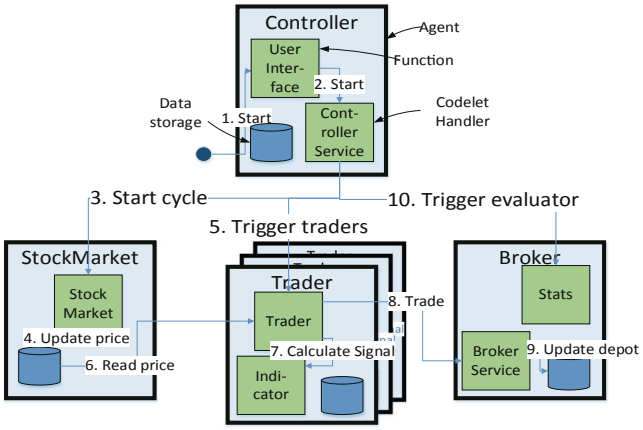

Fig. 5. Design of the stock market simulator

The simulation test was executed on a Dell Precision 3510 laptop with a Mosquitto MQTT server. As the system was running, agents replicated as their deposits grew. To test the upper limit of the agent system, the replication process continued until there were so many agents, that the system stopped working correctly. Figure 6 left shows the results. The duration depends on how many trader agents were instantiated and run in parallel. The duration increases more than linear with the number of agents, probably because more agents need to get data from a single point, the broker. At the same time, the single threaded Mosquitto server has to handle more agents in long lists. The limit was reached after about 9000 steps at around 1670 agents, which manifests in long cycle durations. At that time, $\sim$50000 threads and $\sim$8000 agent functions were running from the ACONA system.

To increase the number of agents, another MQTT broker that runs in more threads and therefore manages more connections that Mosquitto could be chosen. Decoupled functions through MQTT guarantee the modularity, which makes mocking easy. The drawback is that it demands many resources as every function is an own thread and maintains its individual MQTT client. It sets an upper limit to the number of agents that can be used within a multi-agent system. An improvement to increase performance

would be to make two types of functions with the same interface. The first function type of function access the MQTT message bus and can be used for external communication. The other function type only accesses other agent internal functions. It would require that each agent maintain its message bus. Another method to increase capacity is to apply the simulations distributed on multiple computers for parallel execution.

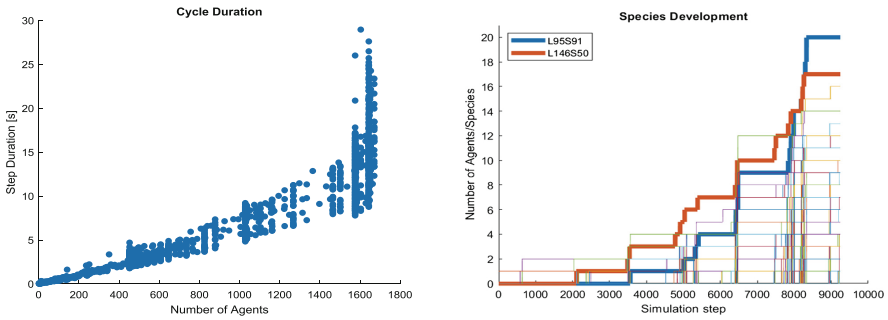

Fig. 6. Left: cycle duration dependency of the number of trader agents; right: development of species per simulation step

The simulation also shows interesting results. In Fig. 6 right, the number of agents per species is shown for each simulation step. The legend shows the most successful species, namely <L146S50> and <L95S91> with 21 respectively 17 individuals. These trading strategies would generate the highest profit. On the other hand, most species with only a few agents. With this setup, 61% of the agents belong to species with less than four agents, i.e., they hardly make any profit but do not lose either.

The stock market game shows that the ACONA framework can be used directly as a multi-agent simulator, where each simulator component is a function or an agent. The potential for evolutionary programming is provided through self-modification during replication. In this application, only parameters were modified. In upcoming applications, thanks to the replaceable functions, also the functions themselves will be the topic of modifications.

4.3 Infrastructure in an Industry 4.0 Application

In the project SAVE (Self-monitoring based process Adaptation for quality assurance in heterogeneous Versatile manufacturing). The simulator part represents a conveyor, which translocate conrods for combustion engines between seven machines. The raw part that has 27 features, such as diameters of the holes and the thickness of a conrod. It passes through seven machines, and each machine modifies one or more of these features. The simulator is a closed loop and implements a model of a machine. The model receives commands that influence how features are generated, e.g., the replacement of a drill results in a smaller hole. The goal of this project/system is to reduce the loss of conrods, which is around 3% at project start. The approach is to add a monitoring and control system called an *Autonomous Cooperative Object* (ACO) to

each machine, which analyzes the features. Together with the ACOs of other machines, it then decides about mitigations as anomalies occur.

The ACONA framework serves as a fast prototyping simulator for the conveyor and as middleware for the modules used within every ACO. Figure 7 shows the setup of the system with one machine and one ACO. To the left, ACONA is used to model the simulator and to the right the ACOs. Through the connectivity of the framework, the real ACO system connects to the simulator without any non-ACONA interfaces.

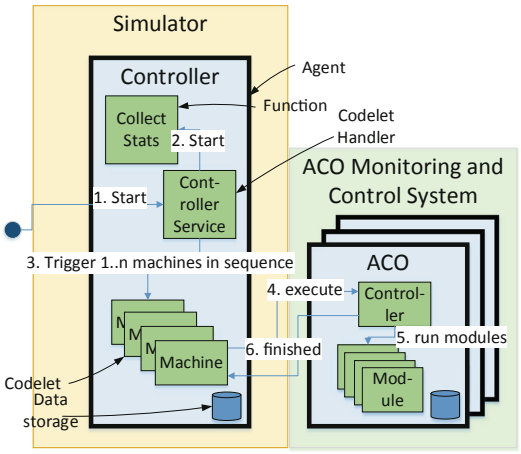

Fig. 7. Design of the conveyor belt simulator and controllers

The simulator is started by a controller that triggers several (1..n) machines in a sequence. Each machine gets the conrod from the data storage of the simulator, modify features through a model and puts the conrod back on another address in the memory. The memory, i.e., the data storage, serves as the conveyor belt. Each machine triggers its associated ACO to run and process measured values. For this application, the whole system runs as a state machine, meaning that the ACO is blocking the simulator while running. This is implemented by using a service call to the codelet handler of the connected ACO. As a result, the ACO sends actions that are applied to the machine model in the next step.

To demonstrate the performance of ACONA, similar to the stock market example, a load test was performed on a Dell Precision 3510. In each cycle, a machine with associated ACO was added. A codelet for statistics collection recorded the number of agents and the cycle duration. As Fig. 8 shows, it was possible to create and add 896 machines with ACOs before the cycle duration started scattering. From about 700 machines and ACOs, the duration for completing a cycle increases much more than linearly. Finally, the system consisted of 8000 running ACONA functions, similar to the stock market example.

This implementation of the first mock system in the project SAVE shows that it is possible to generate a very complex architecture of as well the ACO agent and the

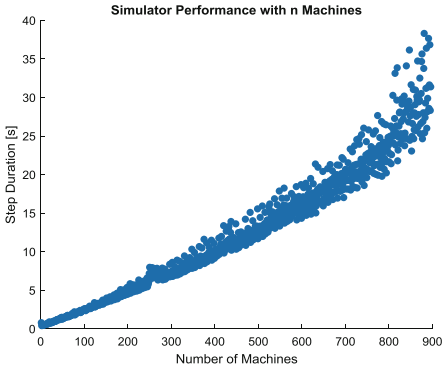

Fig. 8. Cycle duration as a function of the number of machines

simulator for the machines. As the project SAVE is still in the design phase, all software modules were programmed mocks in Java. Eventually, the modules of an ACO will be written in Java, C++, and Python.

5 Conclusion and Future Work

In this paper, we present the agent-based Complex Network Architecture (ACONA) framework. It was initially developed to meet the demands of cognitive architectures in an industrial system, which many other middleware systems can hardly meet it. Agent functions natively offer several common development patterns, reducing the time of having to implement infrastructure. The edge of ACONA in comparison to common middleware is that it provides high flexibility in designing controller systems, simulations, and cognitive architectures. It allows building many types of modular systems and network topologies. Therefore, it qualifies as a middleware in many domains. Through a commonly used communicator, IoT-projects can apply the framework, in many cases without having to write adapters. Another feature that is usually not found in middleware is the ability to apply evolutionary programming in the replication of agents.

The next steps will be to implement clients to allow other programming languages to be used within a single agent. Because functions can be added at runtime, the code is written in other languages than Java will be addable after an agent has started. The potential for further applications of the system is high. In upcoming projects in the Smart Grids domain, the framework will be directly compared to existing solutions. For public use, the ACONA framework is maintained as an open source software at Github[8].

[8] https://github.com/aconaframework/acona.

Acknowledgment. We gratefully acknowledge the financial support provided to us by the BMVIT and FFG (Austrian Research Promotion Agency) program Production of the Future in the SAVE project (864883).

References

1. Kollmann, S., Wilker, S., Meisel, M., Wendt, A., Fotiadis, L., Sauter, T.: Local intelligence for a customer energy management system equipped with smart breakers. In: 2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS), pp. 1–4. IEEE (2017)
2. Weyrich, M., Ebert, C.: Reference architectures for the Internet of Things. IEEE Softw. **33**(1), 112–116 (2016)
3. Wendt, A., Kollmann, S., Siafara, L., Biletskiy, Y.: Usage of cognitive architectures in the development of industrial applications. In: Proceedings of the 10th International Conference on Agents and Artificial Intelligence, ICAART 2018, Portugal (2018)
4. Cejka, S., Hanzlik, A., Plank, A.: A framework for communication and provisioning in an intelligent secondary substation. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–5 (2016)
5. Wendt, A., Faschang, M., Leber, T., Pollhammer, K., Deutsch, T.: Software architecture for a smart grids test facility. In: 39th Annual Conference of the IEEE Industrial Electronics Society, IECON 2013, pp. 7062–7067 (2013)
6. Snaider, J., McCall, R., Franklin, S.: The LIDA framework as a general tool for AGI. In: Schmidhuber, J., Thórisson, K.R., Looks, M. (eds.) AGI 2011. LNCS (LNAI), vol. 6830, pp. 133–142. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22887-2_14
7. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: Mason: a multiagent simulation environment. Simulation **81**(7), 517–527 (2005)
8. Wendt, A., Wilker, S., Meisel, M., Sauter, T.: A multi-agent-based middleware for the development of complex architectures. In: Proceedings of 27th International Symposium on Industrial Electronics 2018 (ISIE), Australia (2018). ISBN: 978-1-5386-3704-3
9. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: Jade — a Java agent development framework. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming. MSASSO, vol. 15, pp. 125–147. Springer, Boston, MA (2005). https://doi.org/10.1007/0-387-26350-0_5
10. Schaat, S., Wendt, A., Kollmann, S., Gelbard, F., Jakubec, M.: Interdisciplinary development and evaluation of cognitive architectures exemplified with the SiMA approach. In: EAPCogSci (2015)