A Multi-Agent-Based Middleware for the Development of Complex Architectures

Alexander Wendt*, Stefan Wilker*, Marcus Meisel * and Thilo Sauter*†
* Institute of Computer Technology,TU Wien, Vienna, Austria
Email: {alexander.wendt, stefan.wilker, marcus.meisel, thilo.sauter}@tuwien.ac.at
† Danube University Krems – Center for Integrated Sensor Systems, 2700 Wiener Neustadt, Austria
thilo.sauter@donau-uni.ac.at

***Abstract* — Complex software systems of today consist of several interacting modules introducing the need of a cohesive architecture. The middleware Java JADE provides this functionality. However, there is a need for additional infrastructure to lower the effort of implementing modular systems. This paper presents the Agent-Based Complex Network Architecture framework (ACONA) that extends the Java JADE multi-agent system by adding a layer, taking away the use-case-based repetitive task of creating a common communication infrastructure, enhanced data storage, and unified function access within the agents. The implementation of a cognitive architecture, which is a highly complex control system in the domain of building automation provides first proof of concept and benchmarking opportunity.**

Keywords—multi-agent-system; cognitive architecture; middleware; ACONA; agent-based complex network architecture framework;

Introduction

A complex software architecture consists of multiple modules that interact with each other. Due to the interactions and relations between the multiple components, the system behavior in general, is hard to model. On the other side, the modeling of individual components may be simple. Therefore, a goal of designing architectures is to break down a complicated architecture with many internal dependencies into a modular, complex architecture, to make it scalable, extendable and testable. For that purpose, usually, middlewares are used. It is a software program that acts as the glue between modules and hides the communication infrastructure. Many middlewares are written for certain domains with their strengths and weaknesses. Such middleware is the agent system Java JADE [1], which is widely spread in the academic world. Agent systems provide scalability because when the problem grows in size; either more agents of the same type can be added to share the load or parts of the problems can be solved in isolation by groups of agents. However, because JADE is very customizable, comfortable infrastructure is missing to make it usable in the implementation of a highly complex system such as a cognitive architecture [2].

In this paper, the middleware Agent-Based Complex Network Architecture (ACONA) framework will be presented. It extends the JADE platform by adding a layer of infrastructure. The research objective is to design a middleware to improve the extendability, testability, and maintainability of JADE-based systems and to lower the developing effort at the writing highly interconnected, modular systems. The strength of the ACONA framework will be conceptually demonstrated as it is used as a base for a cognitive architecture.

Cognitive architectures are well suited as validation as they are "hard" to develop. They model a decision-making process that can be based on psychology or biology. The architectures consist of a large collection of individual modules, which interact according to a defined control logic. Module functionality is mostly executed within the main cycle, but they can also be triggered by events. Most cognitive architectures are designed in a top-down manner although some of them have a mix of top-down controlling and bottom-up concept activation such as object recognition. Further, cognitive architectures process common data structures, where they utilize working memories, adding additional interactions and dependencies. As a comparison of complexity, industrial systems, e.g. in the area of smart grids [3] often consists of separate drivers, a database, and independent algorithms, which control them. Compared to a cognitive architecture, the control flow is straightforward. Therefore, if it would be possible to implement a cognitive architecture, a smart grid industrial application would be to implement a subset of it.

Cognitive Architectures and Frameworks

To understand of the requirements of a middleware that can be used by cognitive architectures, main concepts of this research area described. Cognitive architectures collect input data and select an action to execute from several valid options, which satisfies its internal goals [4]. It bases the selection of action on internal knowledge in the form of implicit knowledge like production rules or explicit knowledge like semantically structured data. Their implementations are general-purpose software, where the same design is supposed to solve several, and different types of problems. Each architecture has developed methods to make decisions. Commonly known architectures are SOAR, ICARUS, and ACT-R, which use a heuristic approach, where a production rule engine triggers production rules. Data is written to a common or separated working memory [2]. Different to the previous ones, SiMA [5] and LIDA [6] origin from models of the human mind. They consist of a chain of modules, which either run serially like in SiMA or parallel in loops like in the LIDA framework.

A possible middleware is the LIDA framework [6]. A scheduler triggers each module that runs at intervals. The inputs activate nodes in a stored graph that contains objects and relationships. Data exchange is done by codelets, which are independent running functions that process data in one memory and put it into other memory modules. Other codelets then regularly poll that memory. That is how data passes through the system while being processed. A drawback is that the LIDA framework only allows loop-based systems. Many applications would prefer event-based systems instead to get more control. While the observer pattern does the data transfer, the architecture is the composition of an XML-file, from which all modules are defined.

To achieve more component independence, the ACNF Cognitive Framework [7] relies on a multi-agent approach. Specialized agents act within a service oriented architecture. It enables scalability of the system. The US Navy has developed a cognitive architecture framework OpenCAF for simulations [8]. However, closer information of the implementation was not available.

In the research area of Smart Grids, due to the nature of heterogeneous hard- and software, middleware is state of the art. A smart grid is an electric grid, where a component of a communication grid optimizes its utilization. Remote algorithms control multiple types of sensors like smart meters and actuators like photovoltaics inverters. Examples of middlewares in this area are Siemens Gridlink [9], TU Wien Demo Facility [3] and OpenMUC [10]. They have in common that they apply a module-based approach and a message-based data transfer. OpenMUC is built on the Java OSGi framework. Each module is compiled as a library and included either at the start or runtime. The data access is organized in channels. Modules write to channels, which are globally available. Data is automatically synchronized between modules through the OpenMUC core. The OSGi engine only allows one instance to run on a Java VM and maximum of 256 modules to be used. Further, only remote debugging is possible, limiting the possibility of making unit tests.

THE FRAMEWORK ARCHITECTURE

The Agent-Based Complex Network Architecture framework (ACONA) provides a framework, where the agents define the nodes of the network and where each agent keeps a set of functions. A multi-agent approach was chosen to overcome scalability problems of frameworks like OpenMUC [10]. Further, agent-based systems implement a strong encapsulation of functions by default, which is a decisive argument. For this implementation, the multi-agent system Java JADE [1] was chosen as a base because it offers a high degree of customizability, is not proprietary and it provides a communication system based on the FIP A protocol. Theoretically, it would be possible to use another message transport system like Apache Kafka or RabbitMQ. ACONA is an additional layer, which extends JADE.

The main idea of the framework is to put as many function dependencies as possible in data structures and not in hard-coded references. By doing that, the internal structure of an agent can be modified at any time. Further, to be able to perform unit tests with low effort, the entire agent shall be instantiable

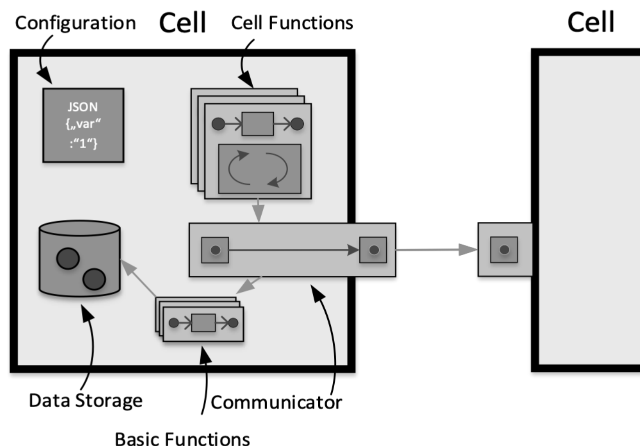

Fig. 1. Top-level architecture of a cell and its five main components

from a configuration string. The configuration can then be adapted for test- or operational- purposes.

In the following, the agent architecture will be described by the components of each agent. The naming of the agent system has been inspired by a biological cell due to its conceptual similarities. Therefore, the software agent is called a *cell*. It consists of several independent functions called *cell functions*, which can either react to external stimuli, process data or act autonomously, e.g. as a loop function. In Figure 1, a top-level view of a cell is shown.

A. Communicator

A developer can save a lot of effort if the entire communication is hidden from the functions. However, JADE does not provide comfortable infrastructure. The developer usually has to put a high effort in the communication. Therefore, a communication module was designed that manages all communication between functions and agents. Among others, there are mainly three ways of letting functions communicate: The most direct way is to use the observer pattern as, e.g. LIDA framework, in which functions are directly linked [6]. It is a fast and easy way of communication, but it causes dependencies between instances. Another method is to communicate through a common memory, where functions put and read their data. While functions within a component are decoupled, two components share the same memory. The third way is through a messaging system, where data is serialized. To the cost of performance, it provides complete decoupling of the functions. To reach complete decoupling, the communicator uses the underlying multi-agent system to communicate with messages between the agents as well as for the internal functions.

Experience shows that it is very time-consuming to create messages and receiver procedures for each type of message in JADE. The communicator provides one basic feature: It allows each function to execute a remote procedure call in any other function in any other agent. For the function, it does not matter in which agent the target function is instantiated. However, to be reachable from outside of the system, each externally available function has to turn on a responder. The advantage of having several functions within an agent is faster messaging and encapsulation of domain functionality.

B. Data Storage

Inspired by cognitive architectures, which all utilize some working memory, each cell needs a container, in which functions can store data structures, which are of interest of other functions. In many cases, a function publishes some data, which is subscribed by other functions. The concept of datapoints provides an elegant way of accessing data through a unique address. JADE only supports a common data storage between sub-behaviors. Therefore a data storage instance with subscription and notification capability is used. It consists of a map, where the datapoint address is put as a key. A datapoint is addressed in the following way: [agent]:[address], where the address can be any string. Here, an example is shown:

```
cognitiveagent1:wm.option.option1
```

JSON-based data structures are used. It allows serialization of many different data types and lowers the effort of the developer. In that way, more data types than the basic data structures String, double, integer and Boolean can be used. The data storage provides the following basic functions: read, write, remove, subscribe and unsubscribe. With these basic functions, all necessary system operations are defined. All functions access the data storage through the communicator. Functions in other agents can subscribe data from a certain datapoint.

C. Cell Function

Cell functions provide all functionalities of the cell. Any cell must have at least one cell function to be able to do anything more than being just a data storage. Often, a cell needs several functions to perform a certain task. While in JADE, any received message triggers all behaviors to start, in ACONA, external messages only trigger published cell functions. It helps to gain better control of internal cell functions. Compared to JADE, function access is standardized to three common access methods, which means less effort for the developer to access the function. Figure 2 shows the three methods a cell function can be accessed.

The first access method is to execute a remote procedure call (RPC Service in Figure 2). It allows other functions to access the function directly through offered methods. Method parameters are passed through a payload object. As the function is blocking, the calling function is blocked as well, while waiting for a response. JADE provides all behaviors as non-blocking, where a scheduler triggers them. In ACONA, functions make extensive use of blocking methods. The reason is that the system communicates through datapoints and it is easier for developers to execute a simple "read datapoint method" with a timeout instead of defining at least two behaviors for each state, first where the read message is sent and then the receipt of the data. The same applies to any other remote procedure calls. Blocking methods come with the cost that more threads per agent are necessary because the functions must not block each other.

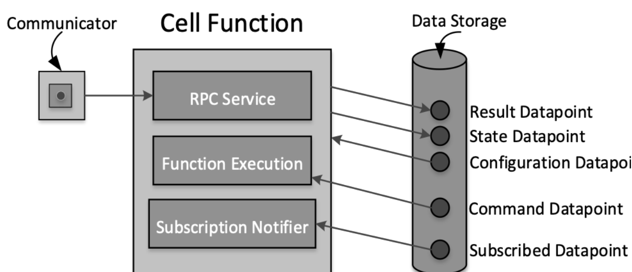

Fig. 2. A cell function with provided access points, datapoints and how the function can be triggered externally

The second method to access a cell function is to directly run the thread. If a service shall not block the caller too long, it can trigger the thread itself by starting the function execution (Function Execution in Figure 2). The *Function Execution* runs like the run method of a thread. It can be accessed by other means as well. Each function provides a datapoint address <command>, where a start, stop and pause commands can be set that control the function execution. To allow both event- and loop-based systems, a function execution can be set to run either once or in a certain interval. While the function execution is running, the datapoint <state> provides any subscriber the current state of the function. It is usable for other functions if they are supposed to wait until the execution has finished.

The third method to access the function is through subscriptions. Every function can subscribe any datapoint in the system. If a subscribed datapoint is modified, the function is notified and can act (Subscription Notifier in Figure 2). Per default, all functions subscribe their own datapoint <command> to be able to be remotely controlled by other functions.

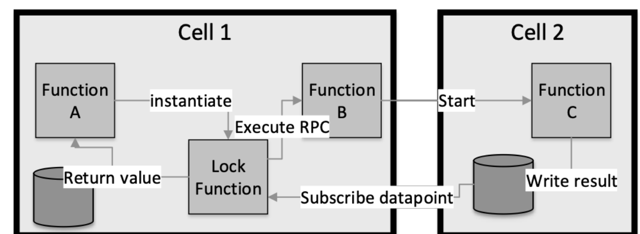

Fig. 3. Control flow of the execution of four functions

The mentioned three possibilities to control a function offer much flexibility as it can be seen in Figure 3. <Function A> instantiates a <Lock Function> with a blocking method. The <Lock Function> executes a remote procedure call in <Function B>. <Function B> starts its function execution, which triggers <Function C>. <Function C> returns a value in its default datapoint <Result>. This datapoint is subscribed by the <Lock function>, which returns the value through a customized method. This setup demonstrated a customized, blocking method. It is useful, when programming controllers, which shall be able to act on events and not blocking the whole system at the same time.

D. Basic Functions

A special type of cell functions are the *basic functions*. They are system functions in each cell, and they provide an additional layer of flexibility. The data storage provide basic data manipulation method like a <read> method. The communicator provides remote procedure calls for cell functions. Because they are cell functions, they can be published and accessed outside the cell, e.g. for a <read> method. Another basic function is the monitoring of the complete system state by subscribing the <state> datapoints of all other cell functions.

E. Configuration

To be able to change function parameter at runtime and to get high testability, the framework can generate cells and their functions completely from a JSON-based configuration. In addition to the LIDA framework or OpenMUC, the configuration of a cell can be made as a file, directly in the code or even within a cell. In that way, the whole configuration of a cell can be received as a datapoint. In the future, the idea is to use this feature to implement evolutionary programming.

To show how the configuration of a cell is made, in Figure 4, an example is presented. The example represents the instantiation of a cell, which implements a weather service. The configuration is generated through a wrapper. The cell uses two cell functions: a weather service function that polls the current weather from a server and a function that records the state of other functions by subscribing its state datapoints. With the method `createAgent()`, a cell configuration `CellConfig` is instantiated. The cell configuration gives the cell a name and optional a class that is instantiated through reflections. For each cell configuration, cell functions can be added with the `addCellFunction()` method. A cell function has a name, here <weatherService>, which must be unique for the cell and a class name <WeatherService.class>. Within the functions, either properties can be set, or managed datapoints can be added. The `setProperty()` method adds a key <WeatherService.CITYNAME> and a value. The key is used inside of the function to get the value. It has proved to be good praxis to use static variables within the function for key names like in Figure 4. The managed datapoints, which are added in the method `addManagedDatapoint()`, automatically subscribe data-points. They can also be triggered to read a datapoint at the start of the function execution or to write a datapoint at the end of the execution. With the `addManagedDatapoint()` method, all external communication is handled. The developer only has to read the key <WeatherService.WeatherAddressID> to get the value inside of the cell function.

```
CellGatewayImpl weatherAgent3 = this.controller.createAgent(
        CellConfig.newConfig(weatherAgent3Name)
        .addCellfunction(CellFunctionConfig.newConfig(weatherservice, WeatherService.class)
                .setProperty(WeatherService.CITYNAME, "stockholm")
                .setProperty(WeatherService.USERID, "tester")
                .addManagedDatapoint(WeatherService.WEATHERADDRESSID,
                        publishAddress , weatherAgent3Name, SyncMode.WRITEONLY))
        .addCellfunction(CellFunctionConfig.newConfig(CFStateGenerator.class)));

weatherAgent3.getCommunicator().write(DatapointBuilder.newDatapoint(weatherservice + ".comma
        .setValue(ControlCommand.START));
```

Fig. 4. Configuration of a cell with two cell functions

Practical Usage of the Framework

The cell with its five components contains all functionality needed for the generation of complex systems, which need a complicated control logic besides of the interaction of multiple sub-components. At the same time, it shall be multi-agent based and allow multiple functions to run independently within a cell.

A. Unit Testing

The focus of the ACONA framework was to provide extended testability. In a complex system, the need to analyze single functions outside of their interactions is of great need. Unit tests and direct debugging are important tools to achieve that. Different to related OSGi-based [10] or Vert.x-based systems [9], where only remote debugging is possible, the ACONA framework provides the possibility to debug functions directly. The developer can create unit tests and launch cells through a JADE agent launcher. From a gateway instance, the developer can access all datapoints and execute all functions of that cell. In the example of Figure 4, the `CellGatewayImpl` <weatherAgent3> is instantiated as a JADE agent. Especially the access to the data storage is useful. Datapoints can be injected into any running cell to trigger functions. In Figure 4, the cell performs the `write()` method and writes a start command to the weather service to start it. The address of the command datapoint is the name of the service appended with ".command".

B. Codelets and Codelet Handlers

In many cases, *codelets* are useful to provide a flexible system. They are independent functions, which run either as daemons performing tasks like in the LIDA framework [6] or they can be independent functions, which are triggered by external events. A codelet handler is designed as a cell function with the task to organize codelets. The *codelet handler* is the controller for starting and synchronizing the registered codelets. Each codelet can be exchanged at any time because they are assigned a codelet handler in the configuration. Because codelets are independent functions triggered by a controller, they are especially useful in cognitive architectures.

Figure 5 shows a codelet handler with registered codelets and a data storage. The workflow of an event-based system is the following: All codelets register in the codelet handler. In the ACONA framework, the codelet handler uses a concept called *run order* to queue the execution of codelets. The codelet handler alone implements a scheduler. All codelets in a run order are executed in parallel. All three codelets of <Run order 1> are executed in parallel, processing some datapoint. Then the single codelet of <Run order 2> is then executed that continue the processing of the datapoint. The run orders are useful if rules are depending on each other.

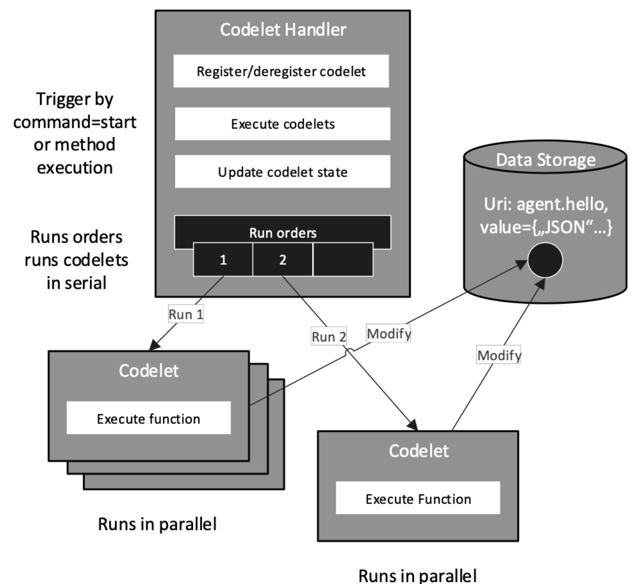

Fig. 5. A codelet handler and connected codelets

C. Performance Tests

The performance tests show how many cells can be connected and used at the same time in serial or parallel connection. All tests were performed on a Dell Precision 3510 with Core i7 and 8GB RAM.

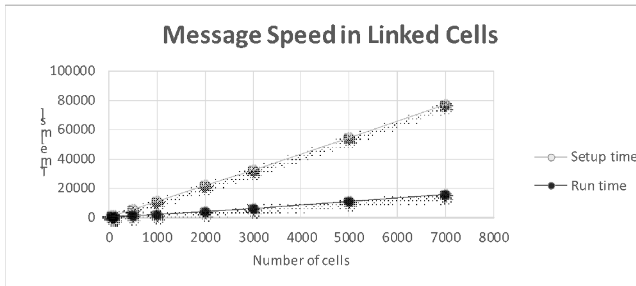

Fig. 6. Cells linked through subscriptions

In the first test, 50 to 7000 cells were ordered in a series to subscribe a datapoint of each other and to pass it to the subscribing neighbor. The result in Figure 6 shows that the system scales linearly to the number of agents. No upper limit was reached. The setup of duration takes about five times longer than the function execution. The reason is that at setup time, in each agent multiple operations are performed serially. An example is to add default datapoints and to set up subscriptions.

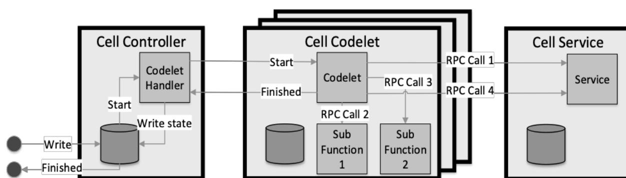

Fig. 7. Performance test of a multi-agent system

In the second test, the scalability of a codelet handler was tested. A codelet handler was instantiated in an agent <Cell Controller> as shown in Figure 7. A service function <Service> that provided two methods was instantiated in the cell <Cell Service>. Up to 2300 codelets were instantiated as cells represented by <Cell Codelet>. Each codelet consisted of three cell functions: One codelet function that acted as a controller for the cell and two business logic functions that performed simple tasks. The codelet function accessed two methods in the cell function <Service> through remote procedure calls.

The codelet handler started all codelet cells as run order 0, i.e. at the same time. Figure 8 shows that both setup and the

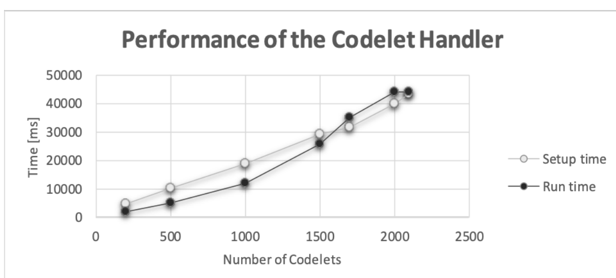

Fig. 8. Setup and execution time of the codelet handler depending on the number of codelets

execution time scales almost linear with the number of codelets up to 2300 codelets. Up to around 1600 codelets per codelet handler, the setup time is higher than the actual execution time. At around 2500 codelets, the system reaches timeouts and reaches the maximum number of codelets per codelet handler. The system is still scalable as it is possible to use multiple codelet handlers, which are controlled by a top codelet handler. The tests show that the framework can handle much more cells than necessary for a cognitive architecture.

Implementation of a Cognitive Architecture

In the project KORE (Cognitive Optimization of Control Strategies for Increasing Energy-efficiency in Buildings), the cognitive model SiMA [5], originally used in the domain of artificial life simulations, was used as inspiration to solve a problem in the area of building automation [11]. The task was to generate control strategies for a building automatically based on semantic building data and previous simulation runs. The goal of a control strategy is to keep an acceptable CO_2 comfort level for each room in the building and at the same time minimize energy consumption while considering certain desired behavior by the system. The task of the cognitive system is to execute all tasks of the system depending on priority and order.

Real world applications of cognitive architectures are rare. It turns out that although cognitive architectures are supposed to be general-purpose software, their implementations are often tailored for intended use. It puts strict requirements on the flexibility and adaptability of the system. The cognitive architecture in KORE implements a general *cognitive process* [4], which is presented within the architecture in Figure 9. It executes the following steps serially: (A) reads system inputs, i.e. sensor data; (B) Classifies and transforms the input to an internal representation; (C) activates system goals based on sensor data; (D) activates beliefs, which are inferred from sensor data, i.e. relationships between perceived objects; (E) proposes options, i.e. the possible actions based on the beliefs and the system goals; (F) propose specific actions for each option; (G) evaluates each option based on goal fulfillment and effort to reach it; (H) selects the option that has the highest score from the evaluation; (I) executes the corresponding action; (J) transforms the action from the internal representation into the language of the system driver. Each process step modifies the *working memory*, which contains perceived data as well as the *internal state memory*, which keeps the state of the goal and options. The cognitive architecture maintains several state machines, represented by options in the internal state memory and because only one state transition can be made per cycle, the system evaluates each option.

The cognitive process is implemented as a top codelet handler within a single cell, where each process step gets another run order. Figure 5 shows how it works. Each process step is also implemented as codelet handlers, six in total. Finally, for each process step, codelets with the data processing in memories is implemented. It has the advantage that the basic architecture remains constant while all problem specific codelets can easily be exchanged and extended. Because all codelets are defined in the configuration file and are invoked through

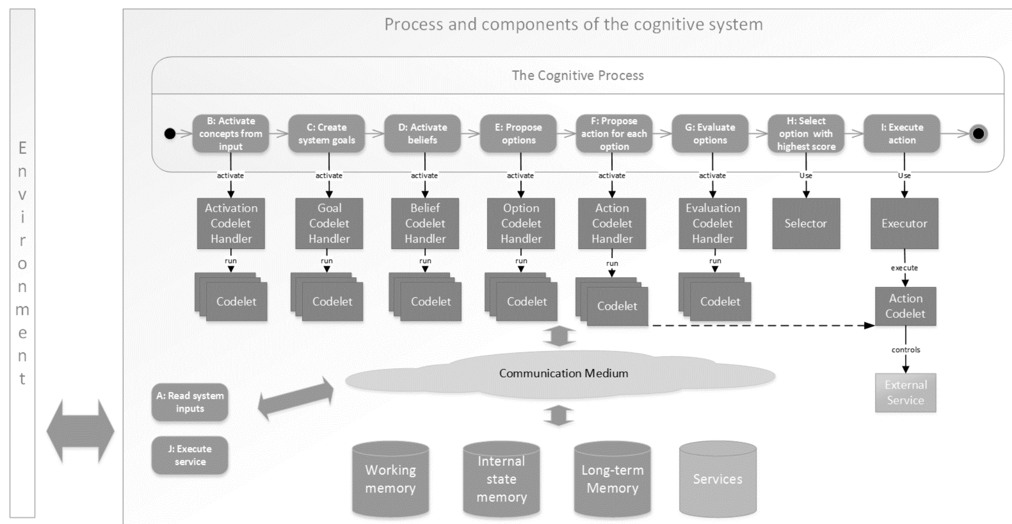

Fig. 9. The cognitive architecture implemented by the ACONA framework an application in the building automation domain

reflections, the features of the architecture can be adapted to other problems with low effort. The working memory and the internal state memory are implemented as namespaces of the knowledge base of the cell. In the current solution, all long-term knowledge is stored externally in an OWL triple store. If there is a need to add a long-term memory directly to the agent, only a new namespace has to be created in the cell. The current implementation applies 37 codelets and around 20 other cell functions utilizing, e.g. SP ARQL drivers, REST server and system actions (services in Figure 9).

Discussion

The ACONA framework has been designed to serve as an infrastructure for complex systems and shall improve extendability, maintainability, and testability. It is an agent-based middleware that extends Java JADE by adding an enhanced data storage, hiding communication and unifying function access. A general-purpose cognitive architecture for the building automation domain was implemented in the ACONA framework. A conceptual comparison with other representative frameworks shows that ACONA allows the necessary flexibility to fulfill the research objectives. Any desired functionality can be implemented on JADE by using this framework. Performance tests show that the framework is scalable. The maximum number of parallel cells accessing a single controller cell was about 2300. However, a top controller can have multiple sub-controllers.

Due to the advantages of flexibility and scalability, ACONA will be applied to smart grid applications for problems, which require more control logic than just connecting sensors with actuators. Another goal of ACONA is to enable evolutionary programming. The foundations have already been set by letting a cell invoke all its functions from a configuration. It can implement many types of functions and control logic. Just as in biology, each cell is supposed to be able to replicate and mutate some of its code during replication. The challenge is to find the suitable problem to solve with self-replicating systems.

The current version of the ACONA framework is available on Github at `https://github.com/aconaframework/acona`.

Acknowledgements

We gratefully acknowledge the financial support provided to us by the BMVIT and FFG (Austrian Research Promotion Agency) under the KORE project (848805).

References

[1] F. Bellifemine, F. Bergenti, G. Caire, und A. Poggi, „JADE—a java agent development framework“, in *Multi-Agent Programming*, Springer, 2005, S. 125–147.

[2] P. Langley, J. E. Laird, und S. Rogers, „Cognitive architectures: Research issues and challenges“, Cogn. Syst. Res., Bd. 10, Nr. 2, S. 141–160, 2009.

[3] A. Wendt, M. Faschang, T. Leber, K. Pollhammer, und T. Deutsch, „Software architecture for a smart grids test facility“, in *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, 2013, S. 7062–7067.

[4] A. Wendt, S. Kollmann, L. Siafara, und Y. Biletskiy, „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.

[5] S. Schaat, A. Wendt, S. Kollmann, F. Gelbard, und M. Jakubec, „Interdisciplinary Development and Evaluation of Cognitive Architectures Exemplified with the SiMA Approach.“, in EAPCogSci, 2015.

[6] J. Snaider, R. McCall, und S. Franklin, „The LIDA framework as a general tool for AGI“, in International Conference on Artificial General Intelligence, 2011, S. 133–142.

[7] J. A. Crowder, J. N. Carbone, und S. A. Friess, Artificial cognition architectures. Springer, 2014.

[8] J. S. Steinman, C. N. Lammers, und M. E. Valinski, „A proposed open cognitive architecture framework“, in Winter Simulation Conference, 2009, S. 1345–1355.

[9] S. Cejka, A. Hanzlik, und A. Plank, „A framework for communication and provisioning in an intelligent secondary substation“, in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, 2016, S. 1–5.

[10] S. Feuerhahn, M. Zillgith, R. Becker, und C. Wittwer, „Implementierung einer offenen Smart Metering Referenzplattform-OpenMUC“, in *ETG-Fachbericht-Internationaler ETG-Kongress 2009*, 2009.

[11] G. Zucker, A. Wendt, L. Siafara, und S. Schaat, „A cognitive architecture for building automation“, in *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, 2016, S. 6919–6924.