1 An Agent-Based Middleware for the Development of Complex Architectures

Within this project, a multi-agent based middleware was developed to enable the implementation of the cognitive model. The software project, called Agent-Based Complex Network Architecture (ACONA) is located on Github on https://github.com/aconaframework/acona.

1.1 The Need for an Agent-based Middleware

A complex software architecture consists of several interacting modules. It makes the system behavior generally difficult to model, due to the interactions and relationships between the components. However, individual components can be easily modeled.

Applications of complex architectures can be found in the area of smart grids [WFLP13], multi-agent systems [BBCP05] and cognitive architectures [DiZu08]. Smart grids combines an electric grid with IT infrastructure, where heterogeneous hardware is connected with algorithms. There are drivers for each device, often written in different languages. There are databases to store that data and finally, algorithms like a tap changer algorithm process the data and defines set points, which are written as commands to the devices. In a multi-agent system, each agent may have a very simple behavior, but as an ensemble the system may show emerging behaviors, due to the interaction of the individual agents. Finally, cognitive architectures mimic the decision-making of living beings. Their purpose is to enable a computer to prioritize and select tasks just as good as a human can do.

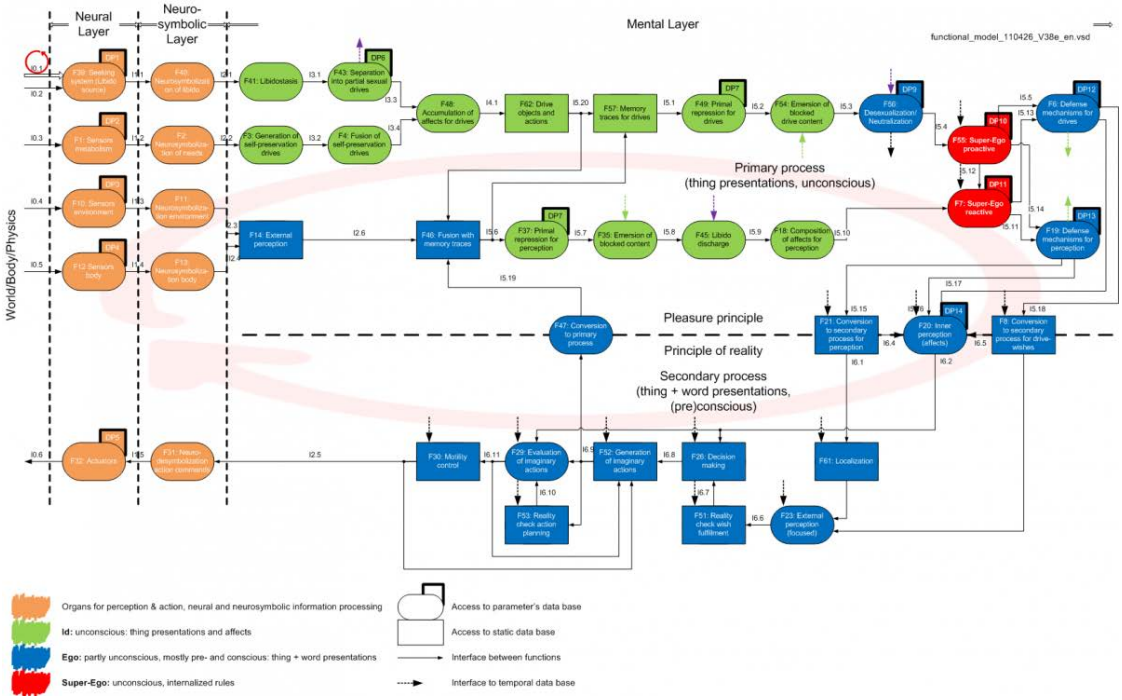

Figure 1-1: Complex system of the SiMA cognitive architecture [SWKG15]

All of the above systems face the developer with complicated problems. The goal is therefore to break down a complicated architecture into a modular, complex architecture. It makes the architecture scalable, extensible and testable. The following requirements apply on a complex system:

- Handle heterogeneity: E.g. several types of hardware from different producers are used in a system
- Handle complexity: Enable interaction between different components
- Handle adaptability: Make the system easily testable on component level and allow integration of new components without having to modify larger parts of the system

Usually, a middleware is used for this purpose. A middleware is a software program that acts as a glue between modules and hides the communication infrastructure. It connects disconnected components through an application. It hides the communication infrastructure from the developer and it provides a common interface for the applications, which are part of the system.

1.2 Goal and Problem Statement

In the research area of smart grids or building automation, there is a demand to implement artificial intelligence in the form of cognitive architectures. Therefore, the goal of the ACONA framework is to enable the implementation cognitive systems in industrial systems on one middleware. Everyone, who has worked with cognitive systems, knows that they are highly complex and “hard to implement”. In Figure 0-1, the model architecture is shown. Around 45 modules are connected and most of them access a database layer too.

Such a middleware has to satisfy some criteria:

Agent system with „body“ and encapsulated internal functions: An agent system is needed, where the agent provides the body for e.g. a cognitive architecture. The agent need internal functions, which are not directly accessible from other externally. For the use of multiple agents within an environment, i.e. a multi-agent-system, the middleware has be able to handle this.

Necessary infrastructure provided by internal functions: Basic patterns like request/response and publish-subscribe shall be available. All inter-function and inter-agent communication shall be hidden from the developer. Functions shall be able to run in cycles or be triggered on events.

Flexible enough to support a cognitive architecture: The idea is to create one client that represents the embodiment of a cognitive architecture and cognitive functions to it. The functions shall be independent, i.e. clients, which exchange messages. As the functions can be small, there should not be a too big overhead in programming and creation of infrastructure to instantiate the function. Each function can be done as a codelet in LIDA [FMDS14] or rule in SOAR [LCCD12] and they only check one thing when they are triggered.

Existing middleware are able to partly fulfill the proposed requirements. Many of them have a proper infrastructure, where they allow the developer to develop modules. However, they miss the functionality of an agent system. Usually, one client is considered to be a single agent. But then the internal functions are missing.

Java JADE [BBCP05] provides a highly customizable multi-agent system. It is built as a multi-agent system, where each agent can implement behaviors. A behavior can be very simple, to do only one thing, but also very complicated, which is suitable for building a cognitive architecture. Each agent provides communication through the FIPA protocol. Therefore, it is suited to be used to build a cognitive architecture. However, JADE only provides a minimal infrastructure for using the behaviors. In a cognitive architecture, many functions should be created from the same template. Therefore, a template function has to be generated that provides all necessary patterns. If not, the system developer has to repeat a lot of programming instead of concentrating on the program logic. Finally, behaviors in JADE are not supposed to be parallelizable, i.e. if one behavior is blocked, the whole agent is blocked. Behaviors are always executed in sequence, where they finish their state transitions and then the next behavior can start. That makes it hard to create a cognitive architecture within one agent, which parts runs in different threads. An example of that problem is found in the cognitive agent SiMA [WGFS15], where there is a primary and a secondary process. In theory, these processes should be able to run in parallel.

The idea is to use Java JADE as a base. The existing agent concept as well as the means of communications are used. It is extended with an additional layer. This layer adds thread-based functions instead of behaviours, function dependencies put into the data instead of direct references and external connections to e.g. Junit, where data can be directly injected into the agent. The result is the **Agent-based COmplex Network Architecture framework (ACONA).**

1.3 The ACONA Model

The main idea of the ACONA model has been inspired by micro-biology. To understand the purpose of the model and its functionality, it is good to keep that in mind. It is tried to mimic the functionality of biological cells. A cell is the atomic unit of living beings. It consists of a membrane to distinguish itself from the environment. Therefore, each agent is called a *cell*. Inside, it keeps encapsulated functions in

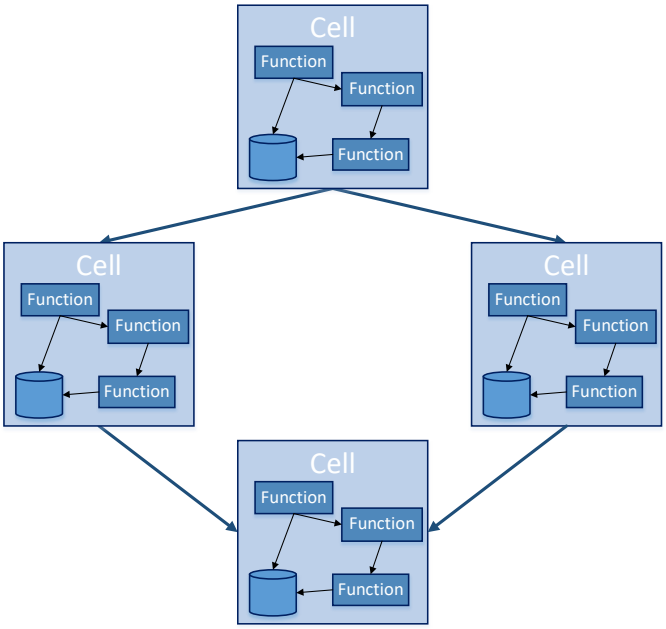

Figure 1-2: Interacting cells

the form of proteins. Each protein fulfils a certain role within the cell. The proteins define the functionality of the cell as a whole. They execute a function if certain substances, i.e. triggers are available. It is a good correspondence to production rules or codelets in software. These functions are called *cell functions*. A special protein in a biological cell is the DNA, which keeps the whole configuration. Proteins are generated by reading the DNA and then synthesizing them in a protein factory. In the same manner, all cell functions of the cell are generated from a configuration.

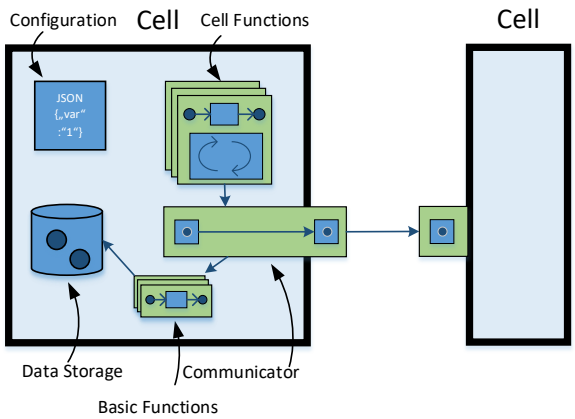

Figure 1-3: High-level model of a cell

1.3.1 The Cell

As can be seen in Figure 4-3, the cell consists of the following components: A communicator, a data storage, cell functions, basic functions and the configuration. In the following, each component is

described in detail. As the ACONA framework is implemented on Java, examples and methods will be shown in Java code as well. Finally, the internal communication between modules is demonstrated.

1.3.2 Communicator

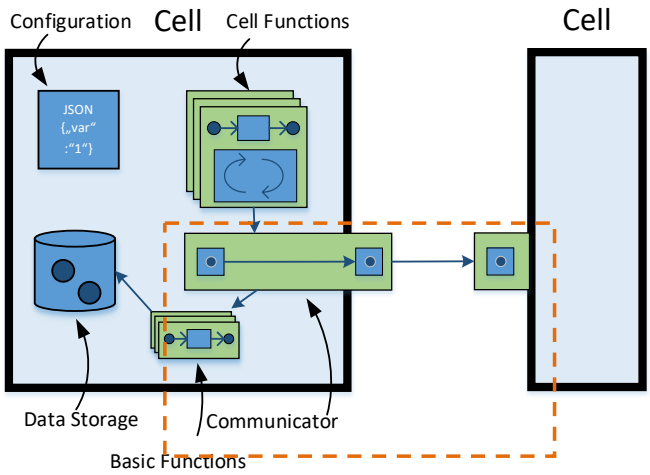

Figure 1-4: Communicator

The communicator manages all communication between Cell Functions of the cell. It hides the whole communication infrastructure of JADE from the user. With the communicator, the designer of a cell function can reach any other function in any other cell if the function is externally reachable (see configuration). The communicator provides one basic action: A Remote Procedure Call (RPC) to another cell function. As shown in Figure 4-4, the communicator has two means of communication: internally by getting the target function either from the CellFunctionHandler or externally by starting a RequestResponseBehavior to another cell. Based on the execute function, all other communication functions are generated. In the following code, an example is shown.

First, a list of data point addresses is created and an address to read is added.

```
List<String> sendList2 = new ArrayList<>();

sendList.add(“someagent:datapointaddress.1.sensorvalue”);
```

The list is added as a parameter to a JsonRpcRequest, which shall execute the method <read> in the target function. It means that the datapoint <someagent:datapointaddress.1.sensorvalue> is the parameter of the method <read> in the target function.

```
JsonRpcRequest request = new JsonRpcRequest("read", 1);

req2.setParameterAsList(0, sendList);
```

The communicator executes the method <execute> with an agent name, service name, the JsonRPCRequest and a timeout for the request. This request is blocking.

```
JsonRpcResponse result = client.getCommunicator().execute(“someagentname”,
"someservicename", request, 100000);
```

If no timeout is provided, the default timeout is used. It can be set any time with the following method.

```
cellControlSubscriber.getCommunicator().setDefaultTimeout(10000);
```

As a response, a JsonRpcResponse is received, with either a result, or an error. In this example, the response would contain the datapoint at the target address.

For basic functions like reading or writing values, default functions have already been defined in the communicator. These are the functions, which modifies data in the common data storage of each cell. They are the core of the ACONA framework. These functions are: read(), readWildcard(), write(), subscribeDatapoint(), unsubscribeDatapoint() and remove().

For the read() function, an example is shown. The read function takes an address [agent:datapointaddress] and returns a datapoint. It is actually doing the same as the method in the previous example. The result is a data point, which always contains a Google Gson JsonElement. The JsonElement can then be converted into any wished structure. In the example below, it is converted into a JsonObject

```
JsonObject result =
this.getCommunicator().read(“someagent:datapointaddress.1.sensorvalue”).
getValue().getAsJsonObject();
```

The function ReadWildCard() is used to read multiple datapoints from a storage. In the following example, all datapoints from the addresses, which start with “agent1:multipledatapoints.channels.” are

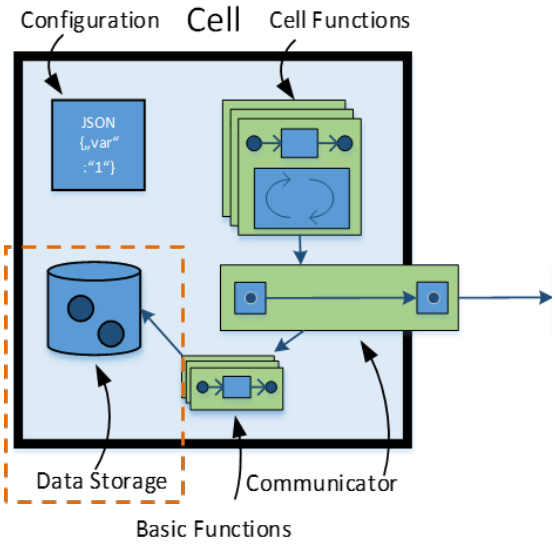

Figure 1-5: Data Storage

read. The “*” is the wildcard.

```
List<Datapoint> result = this.getCommunicator().
readWildcard(“agent1:multipledatapoints.channels.” + "*");
```

1.3.3 Data Storage

Each cell contains a data storage. The idea is that cell functions can share their data through datapoints, which are available through a sort of datapoint server, the data storage. The data storage is a map containing the address of a datapoint as the key and the actual datapoint.

The data storage provides the following methods: read(), readFirst(), write(), append(), remove(), subscribeDatapoint(), unsubscribeDatapoint(). They are almost the same functions, which are available in the communicator through basic functions. Each datapoint can be read, written to, appended to, removed or subscribed. In case of subscription, all subscribing functions are notified on any change of the subscribed datapoint.

Datapoints are a common mean of communication in middleware. Their addresses are defined in the following way: [agent]:[data storage address], e.g.

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

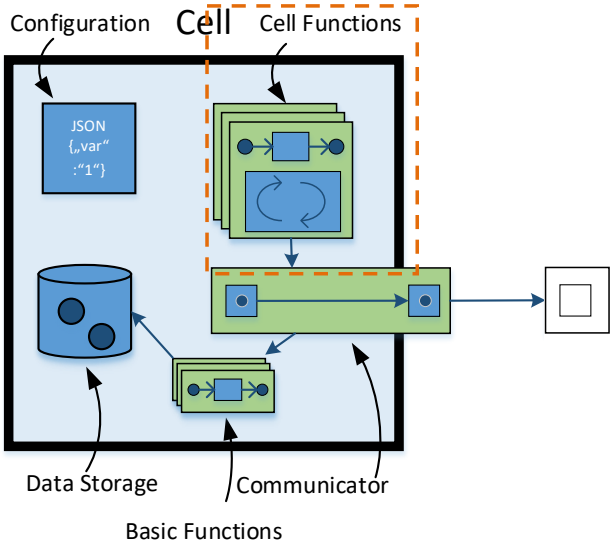

Figure 1-6: Cell function

Their content can be any Json serializable class. Compared to byte array, these values are always readable, which provides additional comfort at the debugging. The datapoint provides Google Gson conversion methods in both directions. For instance, an instance of the class <Episode> can be read in

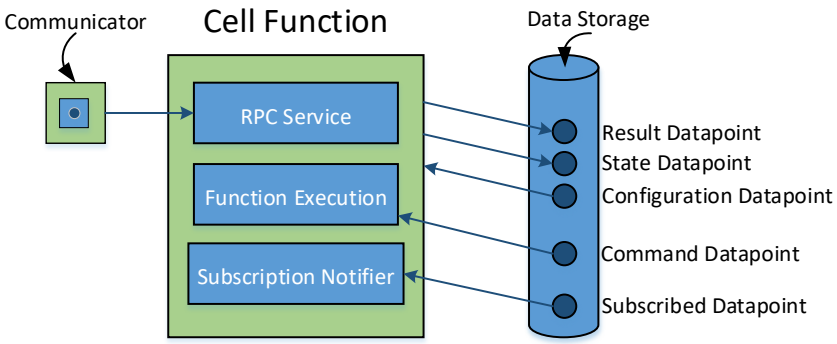

Figure 1-7: Modes of a cell function

this way:

```
Episode result = cogsys.getCommunicator().read("test").getValue(Episode.class);
```

1.3.4 Cell Function

The Cell Function gives the cell its characteristics. Often, a cell needs multiple functions to perform its task. Its purpose is to provide a template, which contains all necessary methods, which are desired in the design of a complex system. Usually, there is a trade-off between flexibility and comfort. If there is a high flexibility, i.e. customizability, then the effort is high for designing any function. As seen in Figure 4-8, the cell function is connected to the communicator, which handles all communication to other cell functions within the cell or to any other cell function in another cell. Cell functions within the cell are managed through a CellFunctionHandler (see Figure 4-4) to which all cell function register.

To be accessible externally, each cell function uses datapoints in its local data storage, see Figure 4-7:

- State: Here, each functions publishes its current state [BUILDING, INITIALIZING, RUNNING, FINISHED, ERROR, UNDEFINED]
- Command: the function can receive a command externally [START, STOP, PAUSE, EXIT], where the <START> and <STOP> controls the internally running thread of the function
- Configuration: Each function can receive new configuration parameters through a datapoint
- Result: If the function shall return a result, this is the default datapoint to do that
- Subscribed: Any datapoint in the system can be subscribed and if a value is changed here, the notification method of the function is started

In the ACONA framework, each cell function can be accessed through three modes: Remote Procedure Call, independent function execution and notifications through subscriptions. An overview is shown in Figure 4-7.

Remote Procedure Calls

The first method that is implemented by a cell function is the

```
public JsonRpcResponse performOperation(JsonRpcRequest param, String caller);
```

The execute() method in the communicator starts this method as a remote procedure call. It is blocking. A JsonRpcRequest contains the method that shall be called and parameters. In the cell function, this method is implemented, where the developer can implement it arbitrary. The method variable in the request from chapter 1.3.2 is therefore optional. If there is no structure handling it, the method is not necessary. In the parameters field, parameters are added. The developer has to decide, whether it is good to use a blocking function here, or if the function shall release and return with an answer through a datapoint or callback.

To make a cell function available to other functions outside of its cell, the following setting has to be set.

```
this.getFunctionConfig().setGenerateReponder(true);
```

It generates an external interface in the cell for that function. Without the external interface, the function is encapsulated within the cell. To understand why the responder function is necessary, please refer to chapter 1.3.7.

Function Execution

The second method of function control is the independent function run. The function is a thread, which runs independently. It offers three methods, which are implemented by the developer:

- executePreProcessing(): Execute collect and read functions before the actual function shall start
- executeFunction(): Execute the main functionality
- executePostProcessing(): Write values from the function, clean up

The function execution can be controlled in several modes. If *execute once* is true, then the function executes just once on trigger.

```
this.setExecuteOnce(true);
```

If *execute once* is false and the *execute rate* is set to e.g. 1500 as below, the function will execute in intervals every 1500ms.

```
this.setExecuteRate(1500);
```

Because some functions subscribe the state of a function, the state is set <FINISHED> after every run, except *finish after single run* has not been set.

```
this.setFinishedAfterSingleRun(false);
```

This is meant to be used for state machines, where the function executes multiple times until it is finished.

The system is controlled by the methods for control.

```
this.setStart();
```

```
this.setStop();
```

```
this.setPause();
```

These commands can also be set externally by writing the commands [START, STOP, PAUSE, EXIT] to the address [agentname]:[servicename].command.

Subscriptions

The third way to access a function is through datapoint subscriptions. Through the subscribe function in the communicator, any data points can be subscribed by the function. There are a couple of ways to handle subscriptions in the system. The first way is directly through the communicator by using the method

```
this.getCommunicator().subscribeDatapoint(address, callingCellfunctionName);
```

The caller function passes its name with the method to be registered at the target system. However, usually the address to be subscribed is not known at design time and has to be injected at runtime or through the configuration. For that purpose, managed datapoints were introduced, where an id is added to the address. Managed datapoints shall help the developer to fast implemented desired datapoint communication without having to think about implementing several communication methods. The address to be subscribed is then the value of a key-value pair. However, if the address shall be added within the function, the following method can be used

```
this.addManagedDatapoint(DatapointConfig.
newConfig("AAA", "agent1:datapointaddress.test", SyncMode.SUBSCRIBEONLY));
```

The id is <AAA> in this case, the address to be subscribed is <agent1:datapointaddress.test> and thy synchronization mode is <SyncMode.SUBSCRIBEONLY>. The synchronization mode <SyncMode.SUBSCRIBEONLY> means that the value is only subscribed. Other synchronized modes are <READONLY>, <READWRITEBACK>, <SUBSCRIBEWRITEBACK>, <WRITEONLY>. In the case of <WRITEONLY>, a value that is written locally to the locally managed datapoints is always written to the datapoint address after the post processing in the function execution. The registration within the function code looks like this:

```
this.addManagedDatapoint(DatapointConfig.
newConfig("AAA", "agent1:datapointaddress.test", SyncMode.WRITEONLY));
```

In the method below, the id <AAA> is received from the value map and the new value is set without having to know the address of the datapoint.

```
this.getValueMap().get("AAA").setValue("any value");
```

Subscribed data then triggers the method updateDatapointsByIdOnThread() to start.

```
protected void updateDatapointsByIdOnThread(Map<String, Datapoint> data);
```

If there is an id available, the key will be the id. Else, the key is the subscribed address.

By using the methods presented here, the developer has several possibilities to customize the implementation to the demands of the project. It allows the development of complex architectures like cognitive architectures.

1.3.5 Basic Cell Functions

Besides of the Cell Functions, there are Basic Cell Functions, which are the default functions of the system. As mentioned in chapter 1.3.2 about the communicator, it executes a remote procedure call. On the other side, therefore a function has to be available to answer on the call. Most of the The Basic

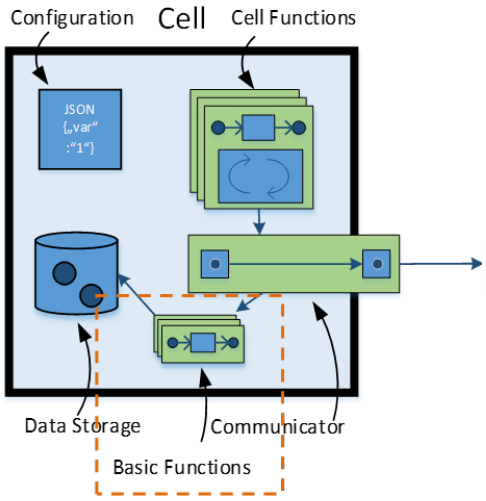

Figure 1-8: Basic cell functions

Cell Functions are Cell Functions that access the data storage. Else, it would not be accessible outside of a cell. These functions are: read(), readWildcard(), write(), subscribeDatapoint(), unsubscribeDatapoint() and remove() and they are integrated in the communicator.

Other Basic Cell Functions are monitoring functions that collects data for the cell about its functions. It is used to find errors or to let a certain function know that a cell has been initialized.

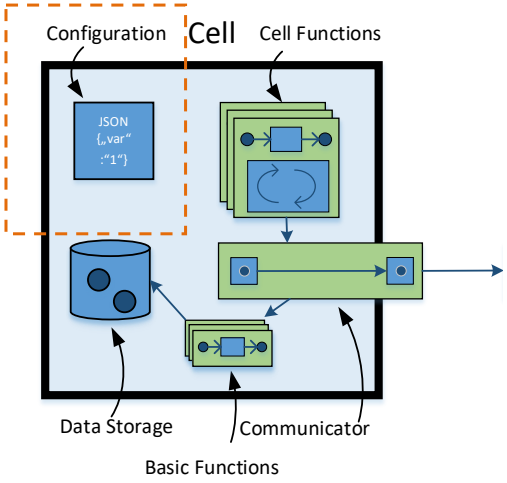

Figure 1-9: High-level model of a cell

1.3.6 Configuration

The configuration is a particular feature in the ACONA framework. The framework has been designed to create completely independent functions. These functions can then be built together within a cell just like LEGO blocks. One big advantage is the injection of mocking functions in the system. Therefore, the whole cell can be generated from a text file with Json syntax.

In the following, an example of an agent with three cell functions is shown. The CellConfig and CellFunctionConfig are wrappers for Json syntax.

```
String weatherAgent1Name = "WeatherAgent1";
String weatherservice = "Weather";
```

```
String publishAddress = "helloworld.currentweather";

CellConfig cf = CellConfig.newConfig(weatherAgent1Name)
       .addCellfunction(CellFunctionConfig.
       newConfig(weatherservice, weatherServiceClientMock.class)
              .addManagedDatapoint(WeatherServiceClientMock.
              WEATHERADDRESSID, publishAddress, weatherAgent1Name,
              SyncMode.WRITEONLY))

       .addCellfunction(CellFunctionConfig.newConfig(CFStateGenerator.class))

       .addCellfunction(CellFunctionConfig.newConfig("LamprosUI",
       UserInterfaceCollector.class)
              .addManagedDatapoint(UserInterfaceCollector.SYSTEMSTATEADDRESSID,
              "systemstate", weatherAgent1Name, SyncMode.SUBSCRIBEONLY)
              .addManagedDatapoint("ui1", publishAddress, weatherAgent1Name,
              SyncMode.SUBSCRIBEONLY));

CellGatewayImpl weatherAgent = this.controller.createAgent(cf);
```

Through the method CellConfig.newConfig() a new, empty cell is generated. The cell could start and it would have its basic functions, but it would not do anything. To add a cell function, the method addCellfunction() is executed on the CellConfig. It allows the generation of a new Cell Function by using CellFunctionConfig.newConfig(). Here, a name of the cell function is provided as well as the class path to the class that shall be instantiated through reflections. Often, the name of a cell function plays a major role in the communication, as the name defines the addresses of the cell functions.

Each cell function can then add different properties. In the example above, managed datapoints are added. For instance, there is an id <ui1> with the address < helloworld.currentweather> and synchronization mode <SUBSCRIBEONLY>.

An important part of the configuration is to set different properties. In the configuration, this is done with key-value-pairs like here

```
.setProperty(MqttAconaClient.PARAMMQTTBROKERADDRESS, mqttBrokerAddress)
```

Best practice has shown the following pattern for implementing any parameter into the functions. Within a cell function, the following variables are created for a property that shall be used

```
public final static String PARAMDATABASEADDRESS = "databaseaddress";
```

The public static string is available as a key for the configuration outside of the function. Then, the actual variable is created.

```
private String databaseAddress;
```

In the init function of the cell function, finally the variable is initialized with the value from the configuration.

```
databaseAddress = this.getFunctionConfig().getProperty(PARAMDATABASEADDRESS);
```

With this type of configuration, it is possible to generate cognitive systems, which contain more than 50 Cell Functions. Just by changing the datapoint subscriptions and addresses, the functions can be generally connected. The system can be changed at any time. Perhaps the most important feature is that everyone, who is working on the system can develop independent of other functionalities as everything can be mocked.

1.3.7 Detailed Cell Internal Communication

The cell is the glue that keeps the system together. In Figure 4-4, the cell is described in more detail. The communication between two functions will be described with an example of how a serialized value is read by a function in <Cell1> from a data storage in <Cell2> at the address <my.address>.

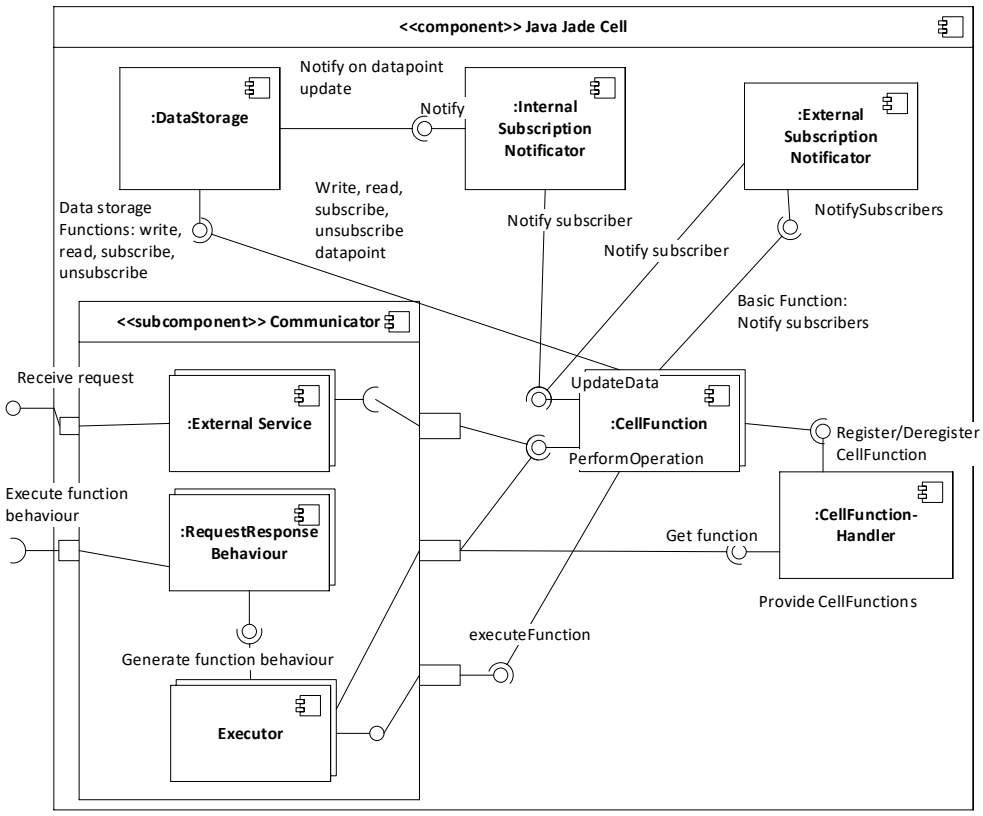

Figure 1-10: Component Diagram of a cell

For the basic cell function <read>, a shortcut has been added to the communicator. In the Java code, the read method is called in the following way:

```
SerializedClass value =
this.getCommunicator().read("Cell2:my.address").getValue(SerializedClass.class);
```

The presented code does the following in the system. First, a CellFunction generates a remote procedure call by using the native `executeFunction` method in the communicator. In case a local remote procedure call would be made, the Executor would get the function instance from the CellFunction Handler and execute the remote procedure call. For a remote call to another agent, a Jade RequestResponse Behaviour is created. In the RequestResponse Behaviour, a FIPA request message is generated and sent to <Cell2>. In <Cell2>, the <read> function has an activated responder, which is represented by the External Service. It executes the offered method <readValue> in the connected CellFunction <read>. The <read> function directly accesses the data storage of <Cell2> and reads the datapoint. The datapoint is then forwarded to the External Service, which returns a FIPA response message containing the requested datapoint to the calling RequestResponse Behaviour. In the Datapoint class, in the getValue() method, a Json parser (Google Gson) is included, which converts the value of the requested class <SerializedClass>.

A subscription of a value from another agent is done in an equivalent way. The difference is that the CellFunction <Subscribe> is connected to an External Subscription Handler, where all agents are registered, which subscribes values. On change of value, a function <Notificator> is triggered that notifies the subscribing agent.

In Figure 4-2, a sequence diagram is shown on how the communication is done between an external agent and the data storage. The external agent (function within an agent) sends a remote procedure call with service name, method name and parameters. In the receiver function service the target service is searched for. If it has been found, the input is validated. According to the FIPA protocol, and agree or refuse is sent back to the calling agent. Refuse is sent if the input could not be validated. If the input was validated, an agree is sent. Then the operation is performed by the cell function. In this particular case, it is a basic function that read, writes, subscribes or unsubscribes a datapoint from the data storage. When it is finished, the result is passed back to the receiver function service that finally returns that value to the caller.

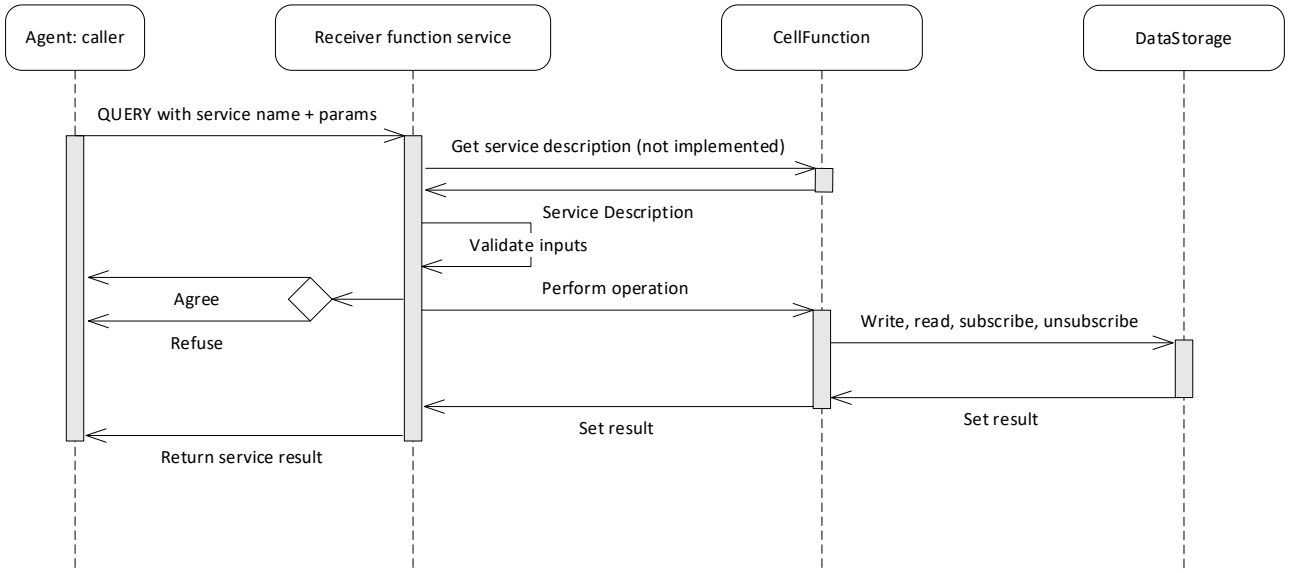

Figure 1-11: Sequence diagram of the access to the data storage from another cell

1.4 ACONA Concepts: Codelets and Codelet Handlers

The ACONA framework provides the base to develop cognitive systems and eventually also to do evolutionary programming. Based on the basic concept of a cell, additional higher concepts were developed. The first concept aims to create a way to execute production rules or codelets. A codelet is here defined as a small piece of code that runs in the system if certain conditions are fulfilled. A typical use is within a cognitive architecture, where a codelet has the task to look for certain objects like a tree in the environment. If a tree is found, an internal representation of that tree is loaded. Each codelet needs to run at a certain time and may have to be synchronized with other codelets. Therefore, a codelet handler is necessary. A codelet handler is the controller of the codelets and executes them on trigger. If the system shall represent a simulator with agents, each agent can be implemented as a codelet and being triggered to execute by the codelet handler.

In Figure 4-12, a codelet handler and its codelets are shown. Both the codelet handler and each codelet are cell functions. The codelet handler provides the following methods, which can be accessed through remote procedure calls:

- RegisterCodelet(): Registers a codelet in the codelet handler with the parameter executionOrder, which tells the codelet handler when the codelet shall start
- DeregisterCodelet(): Deregisters a codelet
- ExecuteCodeletHandler(): Starts the codelet handler, where it runs through all execution orders

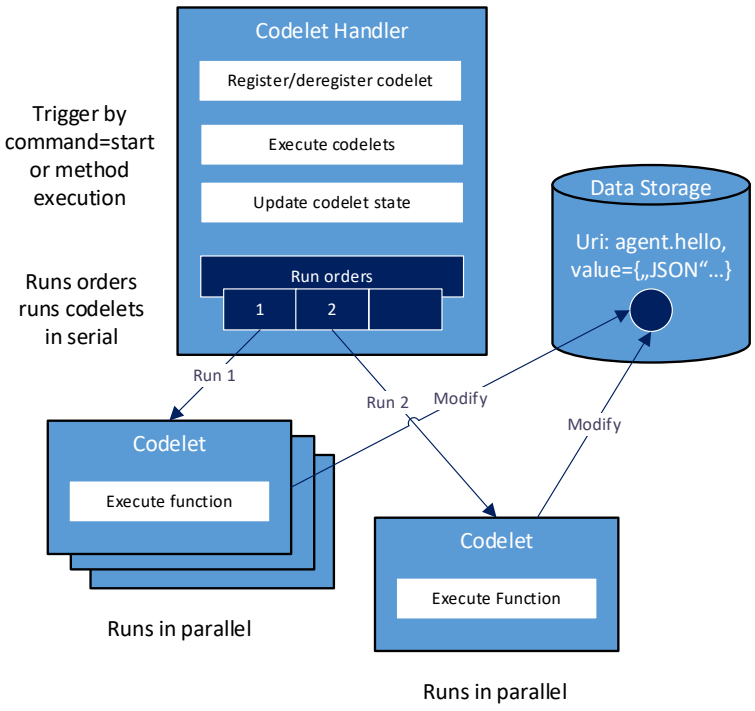

Figure 1-12: A codelet handler and codelets

- SetState(): Sets the current state of the codelets, where all codelets have to be finished for the codelet handler to be finished

The codelet only provides one method that the codelet handler can access through a remote procedure call: startCodelet().

A codelet registers in the codelet handler as the first action. There, it registers its run order. The run order tells the codelet handler when to execute the codelet. The codelet handler starts with the lowest

run order and executes all codelets, which are registered there in parallel. When all these codelets are finished, i.e. have set their states <FINISHED> in the codelet handler, the next run order starts. After all run orders are finished, the codelet handler is finished.

It is also possible to use a codelet handler as a codelet. For instance, if there is a process with modules that shall be executed in serial, where each module consists of codelets, a top codelet handler can control the process modules. Each of the modules, which are also codelet handlers, then executes their codelets on trigger.

1.5 Installation

The ACONA framework can be found on Github at the address https://github.com/aconaframework/acona.

Pull the ACONA framework from https://github.com/aconaframework/acona.git to a location of your choice.

In e.g. Eclipse, selet File-Import and import the ACONA project as a gradle project.

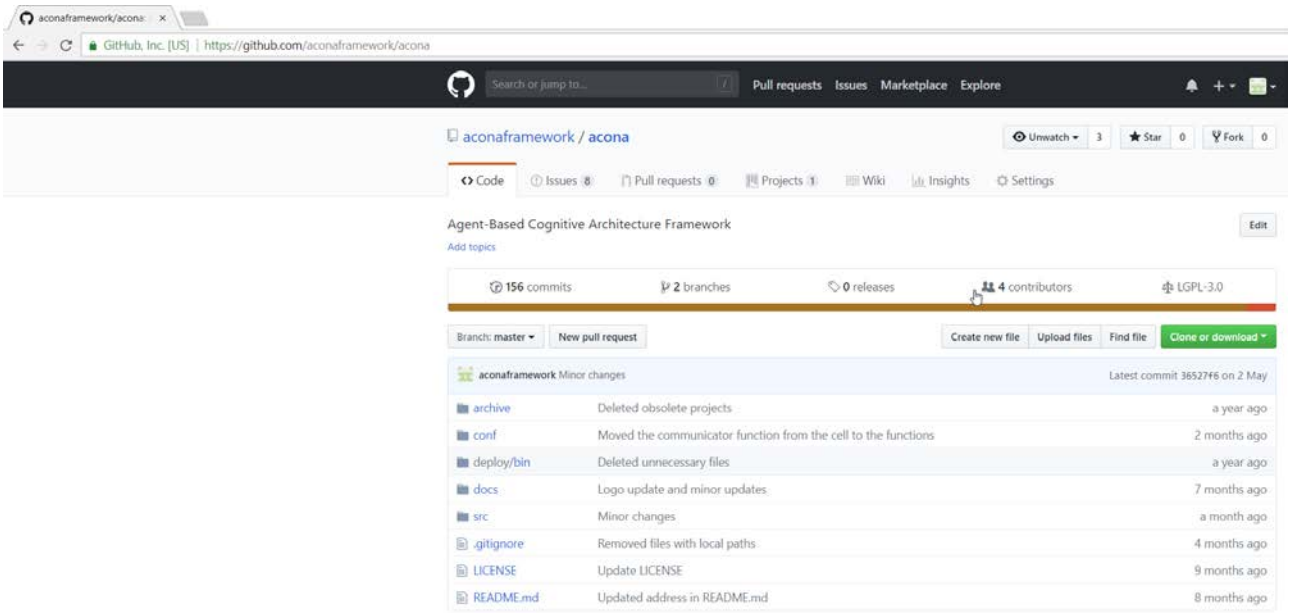

Figure 1-14: Acona at Github

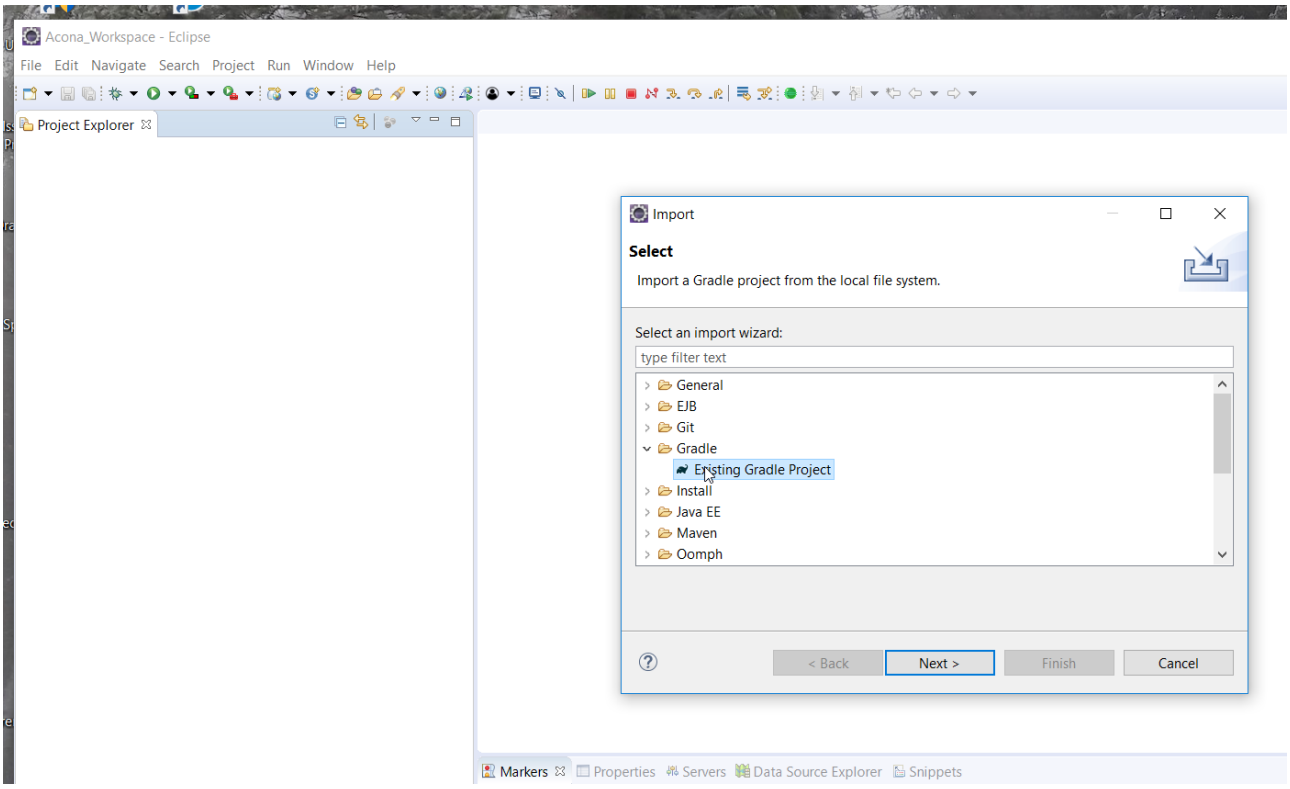

Figure 1-13: Import ACONA in Eclipse

Select each project separately. As the project acona-cell is the main project, it is recommended to start with it. acona-cell has no internal dependencies and all libraries are downloaded by gradle.

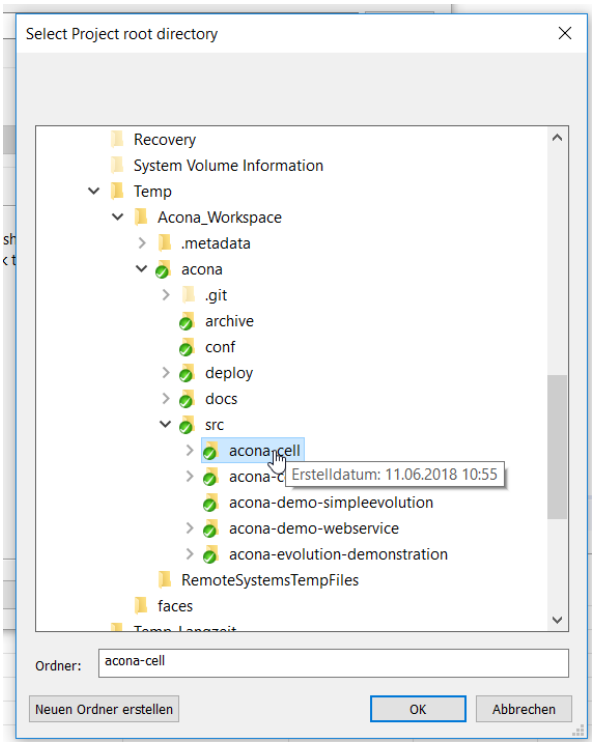

Figure 1-15: Select the acona-cell project

The following projects are available:

acona-cell: The main project for the ACONA framework

acona-cognitiveframework: The cognitive framework, which implements the cognitive process based on acona-cell

acona-demo-simple evolution: Empty experimental project

acona-demo-webservice: Demonstration of how to make a webservice with Acona cell.

acona-evolution-demonstration: Work in progress project of a multi-agent system for the stock market.

To get the dependent projects running, in e.g. acona-cognitiveframework, the settings.gradle has to be changed, to get the right path of acona-cell. In this example, the line `project(':acona-cell').projectDir = new File('C:/Temp/Acona_Workspace/acona/src/acona-cell')` was changed to the absolute location of acona-cell

In the project *acona-evolution-demonstration*, a library called simple-user-console is used. It can be downloaded here https://github.com/aconaframework/commonutils.git

Finally, the functionality of the ACONA framework can be verified through unit tests. All important functionality in the project is covered by unit tests. The tests can be started individually or together by selecting the project->Run as->JUnit Test. It executes all tests of the project.

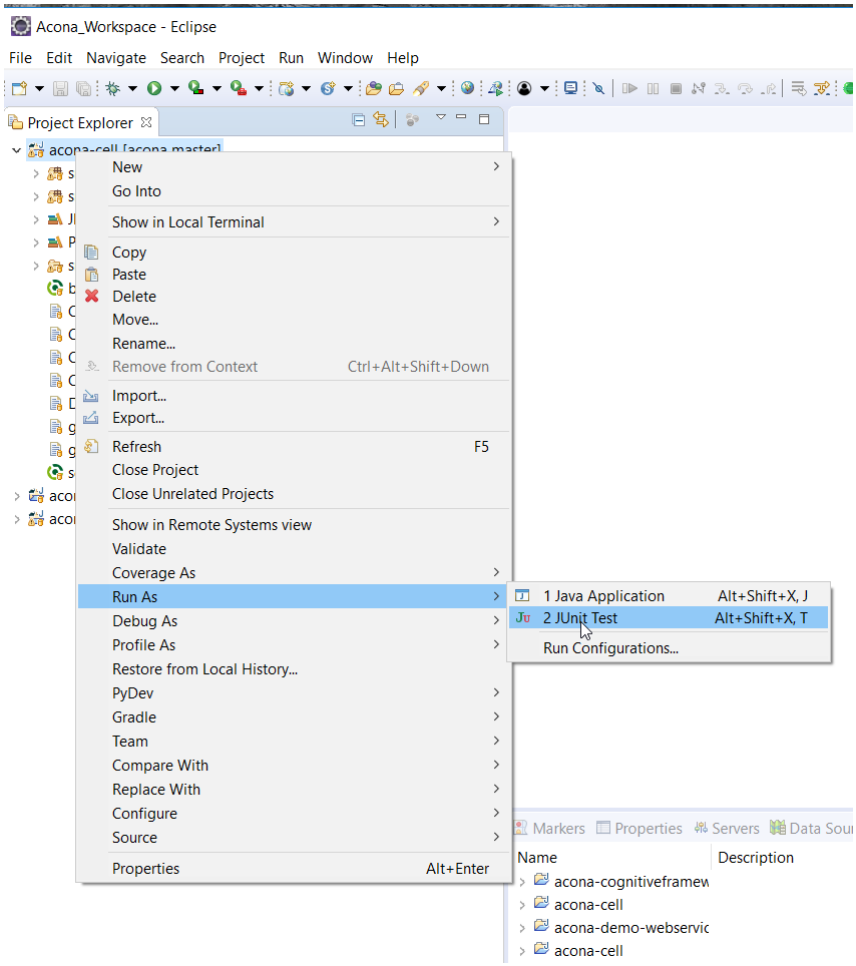

Figure 1-16: Execute Junit tests to verify that all functionality is running

1.6 Tutorial – Writing a Unit Test for Codelet Testing

To provide a good starting point with the ACONA framework, the following tutorials shall demonstrate the usage of the framework by simple implementations.

Testability is a very important issue of modular, complex systems. It is hard to test the whole system at once. Therefore, each module has to be testable. The ACONA framework offers easy connection to JUnit for unit testing of modules. The first step is to setup the unit test. For an executable class, most of the content of the unit test can be copied.

In this test, the idea is to test codelets. A codelet handler and three codelets shall be instantiated in an agent. Each of the codelets shall increment a number that has a certain value. The whole system shall be configured in that way that a number shall be incremented from 1 to 4 by executing the codelet handler three times. The test is passed if the number in the storage has the value 4.

All code for this tutorial can be found here:

/acona-cell/src/test/java/at/tuwien/ict/acona/cell/core/cellfunction/codelets/Codelettester.java

/acona-cell/src/test/java/at/tuwien/ict/acona/cell/core/cellfunction/codelets/helpers/IncrementOnConditionCodelet.java

First, create a new junit class called Codelettester. Then add a logger. Here, the SL4J logger is used with the LogBack implementation

```
private static Logger log = LoggerFactory.getLogger(Codelettester.class);
```

Create the ACONA launcher, which is used to launch the Java JADE environment and to generate cells.

```
private SystemControllerImpl launcher = SystemControllerImpl.getLauncher();
```

In the Setup method, create a main container and a sub container through the launcher.

```
@Before
public void setUp() throws Exception {
	try {
		// Create container
		log.debug("Create or get main container");
		this.launcher.createMainContainer("localhost", 1099,
"MainContainer");

		log.debug("Create subcontainer");
		this.launcher.createSubContainer("localhost", 1099, "Subcontainer");
	} catch (Exception e) {
		log.error("Cannot initialize test environment", e);
	}

}
```

In the teardown method of JUnit, let the launcher stop the system.

```
@After
public void tearDown() throws Exception {
	synchronized (this) {
		try {
			this.wait(2000);
		} catch (InterruptedException e) {

		}
	}

	this.launcher.stopSystem();

	synchronized (this) {
		try {
			this.wait(2000);
		} catch (InterruptedException e) {

		}
	}

}
```

Then, the test function can be implemented. It is called <IncrementOnConditionCodelet.java>. As learnt from best practice in chapter 1.3.6, the variables that shall be passed through the configuration are put as member variables.

```
public static final String attributeCheckAddress = "checkaddress";

public static final String attributeConditionValue = "checkvalue";

private String checkAddress = "";
```

```
private int conditionValue = -1;
```

In the method cellFunctionCodeletInit(), pass the values from the configuration to the local variables.

```
this.checkAddress = this.getFunctionConfig().getProperty(attributeCheckAddress);

this.conditionValue =
Integer.valueOf(this.getFunctionConfig().getProperty(attributeConditionValue));
```

Now, the executeFunction() method is created to fulfill the task of incrementing the value on <attributeCheckAddress> if its value is higher than the value on <attributeConditionValue>. The function content will look like this.

```
//Read address
int value = 0;
String checkValue =
this.getCommunicator().read(checkAddress).getValue().toString();
log.debug("Read value={}", checkValue);
if (checkValue.equals("{}") == false) {
      value = this.getCommunicator().read(checkAddress).getValue().getAsInt();
}

if (value == conditionValue) {
      log.info("Value={} matched. Increment it by 1", value);
      int newValue = value + 1;
      this.getCommunicator().write(DatapointBuilder.newDatapoint(this.checkAddress
).setValue(new JsonPrimitive(newValue)));
} else {
      log.info("Value={} does not match the condition value={}.", value,
conditionValue);
}
```

Notice that the first action is to load the current value from <attributeCheckAddress> and to compare it with the <conditionValue>. If the condition matches, the value is increamented and written back to the same place again. Instead of doing this directly, managed datapoints from chapter 1.3.4 could be used as well. The proper synchronization mode would be <READWRITEBACK>.

Back to the JUnit test file, the necessary parameters of the system are defined. They are the names of the services for the codelet handler and the codelets, the starting value, the expected value and the name of the datapoint, where the value can be found.

```
String codeletName1 = "CodeletIncrement1"; // The same name for all services

String codeletName2 = "CodeletIncrement2";

String codeletName3 = "CodeletIncrement3";

String handlerName = "CodeletHandler";

String controllerAgentName = "CodeletExecutorAgent";

String processDatapoint = "workingmemory.changeme";

// values

double startValue = 1;

int expectedResult = 4;
```

Add an agent with a codelet handler and three codelets. Each codelet is connected to the codelet handler through the variable <ATTRIBUTECODELETHANDLERADDRESS>. Because the codelets are processing the value in the storage, they receive different run orders in the variable <ATTRIBUTEEXECUTIONORDER>. Also the properties <attributeCheckAddress> and

<conditionValue> are defined. While <attributeCheckAddress> is all the same, the <conditionValue> shall be increased by each codelet, i.e. 1, 2, and 3 shall be the values.

```
// Agent with handler and 3 codelets
CellConfig codeletAgentConfig = CellConfig.newConfig(controllerAgentName)
	.addCellfunction(CellFunctionConfig.newConfig(handlerName,
CellFunctionCodeletHandler.class))
	.addCellfunction(CellFunctionConfig.newConfig(codeletName1,
	IncrementOnConditionCodelet.class)

	.setProperty(IncrementOnConditionCodelet.ATTRIBUTECODELETHANDLERADDRESS,
			controllerAgentName + ":" + handlerName)
		.setProperty(IncrementOnConditionCodelet.ATTRIBUTEEXECUTIONORDER, 0)
		.setProperty(IncrementOnConditionCodelet.attributeCheckAddress,
		processDatapoint)
		.setProperty(IncrementOnConditionCodelet.attributeConditionValue, new
		JsonPrimitive(1)))
	.addCellfunction(CellFunctionConfig.newConfig(codeletName2,
	IncrementOnConditionCodelet.class)
...
...
...
CellGatewayImpl controller = this.launcher.createAgent(codeletAgentConfig);
```

At the moment, the createAgent() method is not blocking. Therefore a pause has to be added to give the agent some time to initialize all functions before the test starts.

```
synchronized (this) {
	try {
		this.wait(500);
	} catch (InterruptedException e) {

	}
}
```

Then, it's time to inject the starting value into the address. It is possible to access all data of the cells through the CellGateway. It makes unit testing pretty simple as any state can be set externally for debugging purposes.

```
controller.getCommunicator().write(DatapointBuilder.newDatapoint(processDatapoint)
.setValue(new JsonPrimitive(startValue)));
```

To start the whole system, an execute request is sent to the codelet handler in the same manner as explained in chapter 1.3.2. This method is then executed three times, to let each codelet get triggered and act on it.

```
JsonRpcRequest request2 = new JsonRpcRequest("executecodelethandler", 1);
request2.setParameterAsValue(0, false);

controller.getCommunicator().executeServiceQueryDatapoints(
controllerAgentName, handlerName, request2,
controllerAgentName, handlerName + ".state",
new JsonPrimitive(ServiceState.FINISHED.toString()),
20000);
```

To control the codelet handler, the following method was used: executeServiceQueryDatapoints(). This is a blocking method, which executes a method in a cell function and then subscribes a certain datapoint

```
19:18:57.898 [CodeletExecutorAgent#CodeletHandler] DEBUG at.tuwien.ict.acona.cell.cellfunction.CellFunctionThreadImpl - CodeletHandler>Service
19:18:57.898 [CodeletExecutorAgent#CodeletHandler] DEBUG at.tuwien.ict.acona.cell.cellfunction.CellFunctionThreadImpl - Codelet CodeletHandler
19:18:57.898 [main] INFO at.tuwien.ict.acona.cell.core.cellfunction.codelets.Codelettester - Value is=4
19:18:57.898 [pool-1-thread-4] DEBUG at.tuwien.ict.acona.cell.communicator.AgentCommunicatorImpl - Execute local function=notify, parameters=,
19:18:57.898 [pool-2-thread-4] DEBUG at.tuwien.ict.acona.cell.communicator.SubscriptionHandlerImpl - Activation dp=CodeletHandler.command:"PAU
19:18:57.898 [main] DEBUG at.tuwien.ict.acona.cell.communicator.AgentCommunicatorImpl - Execute local function=read, parameters=, method=read,
19:18:57.898 [pool-2-thread-4] DEBUG at.tuwien.ict.acona.cell.cellfunction.CellFunctionThreadImpl - Codelet CodeletHandler: command PAUSE set
19:18:57.898 [main] DEBUG at.tuwien.ict.acona.cell.core.cellfunction.codelets.Codelettester - correct value=4, actual value=4.0
19:18:57.898 [main] INFO at.tuwien.ict.acona.cell.core.cellfunction.codelets.Codelettester - Test passed
19:19:01.904 [main] INFO at.tuwien.ict.acona.launcher.SystemControllerImpl - Stopping system
19:19:01.905 [main] DEBUG at.tuwien.ict.acona.launcher.SystemControllerImpl - Take down cell=CodeletExecutorAgent@192.168.56.1:1099/JADE
```

Figure 3-17: Console output of the unit test for testing three codelets and a codelet handler

in the system. In this case, the state of the codelet handler was subscribed with a timeout of 20s.

After the tests have run, the actual value is checked.

```
double result = controller.getCommunicator().read(processDatapoint).
getValue().getAsInt();
```

With an assertion, it is checked that the excepted value is equal to the actual value.

```
assertEquals(expectedResult, result, 0.0);
```

If the test was successful, the result of the console would look like this.

References

[BBCP05] BELLIFEMINE, FABIO ; BERGENTI, FEDERICO ; CAIRE, GIOVANNI ; POGGI, AGOSTINO: JADE—a java agent development framework. In: *Multi-Agent Programming* : Springer, 2005, S. 125–147

[DiZu08] DIETRICH, DIETMAR ; ZUCKER, GERHARD: New approach for controlling complex processes. An introduction to the 5 th generation of AI. In: *Human System Interactions, 2008 Conference on* : IEEE, 2008, S. 12–17

[FMDS14] FRANKLIN, STAN ; MADL, TAMAS ; D'MELLO, SIDNEY ; SNAIDER, JAVIER: LIDA: A systems-level architecture for cognition, emotion, and learning. In: *IEEE Transactions on Autonomous Mental Development* Bd. 6 (2014), Nr. 1, S. 19–41

[KoGT16] KOTSERUBA, IULIIA ; GONZALEZ, OSCAR J AVELLA ; TSOTSOS, JOHN K: A Review of 40 Years of Cognitive Architecture Research: Focus on Perception, Attention, Learning and Applications. In: *arXiv preprint arXiv:1610.08602* (2016)

[LaLR09] LANGLEY, PAT ; LAIRD, JOHN E ; ROGERS, SETH: Cognitive architectures: Research issues and challenges. In: *Cognitive Systems Research* Bd. 10 (2009), Nr. 2, S. 141–160

[LCCD12] LAIRD, JOHN E ; CONGDON, CLARE B ; COULTER, KJ ; DERBINSKY, N ; XU, J: The Soar User's Manual Version 9.3. 2. In: *Computer Science and Engineering Department. University of Michigan. Unpublished manuscript* (2012)

[SWKF17] SIAFARA, LYDIA ; WENDT, ALEXANDER ; KOLLMANN, STEFAN ; FERNBACH, ANDREAS ; PREYSER, FRANZ ; KASTNER, WOLFGANG: Automated Generation and Optimization of Control Strategies for Increased Energy Efficiency in Buildings. In: . Austria, 2017

[SWKG15] SCHAAT, SAMER ; WENDT, ALEXANDER ; KOLLMANN, STEFAN ; GELBARD, FRIEDRICH ; JAKUBEC, MATTHIAS: Interdisciplinary Development and Evaluation of Cognitive Architectures Exemplified with the SiMA Approach. In: *EAPCogSci*, 2015

[THHT13] TRAFTON, GREG ; HIATT, LAURA ; HARRISON, ANTHONY ; TAMBORELLO, FRANK ; KHEMLANI, SANGEET ; SCHULTZ, ALAN: Act-r/e: An embodied cognitive architecture for human-robot interaction. In: *Journal of Human-Robot Interaction* Bd. 2 (2013), Nr. 1, S. 30–55

[WDMB12] WENDT, ALEXANDER ; DÖNZ, BENJAMIN ; MANTLER, STEPHAN ; BRUCKNER, DIETMAR ; MIKULA, ALEXANDER: Evaluation of Database Technologies for Usage in Dynamic Data Models-A Comparison of Relational, Document Oriented and Graph Oriented Data Models. In: *ICAART (1)*, 2012, S. 219–224

[WeSa16] WENDT, ALEXANDER ; SAUTER, THILO: Agent-Based cognitive architecture framework implementation of complex systems within a multi-agent framework. In: *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on* : IEEE, 2016, S. 1–4

[WFLP13] WENDT, ALEXANDER ; FASCHANG, MARIO ; LEBER, THOMAS ; POLLHAMMER, KLAUS ; DEUTSCH, TOBIAS: Software architecture for a smart grids test facility. In: *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE* : IEEE, 2013, S. 7062–7067

[WGFS15] WENDT, ALEXANDER ; GELBARD, FRIEDRICH ; FITTNER, MARTIN ; SCHAAT, SAMER ; JAKUBEC, MATTHIAS ; BRANDSTÄTTER, CHRISTIAN ; KOLLMANN, STEFAN: Decision-making in the cognitive architecture SiMA. In: *Technologies and Applications of Artificial Intelligence (TAAI), 2015 Conference on* : IEEE, 2015, S. 330–335

[WKSB18] WENDT, ALEXANDER ; KOLLMANN, STEFAN ; SIAFARA, LYDIA ; BILETSKIY, YEVGEN: 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

[ZSKW18] ZUCKER, G. ; SPORR, A. ; KOLLMANN, S. ; WENDT, A. ; CHAIDO, L. SIAFARA ; FERNBACH, A.: A Cognitive System Architecture for Building Energy Management. In: *IEEE Transactions on Industrial Informatics* Bd. 14 (2018), Nr. 6, S. 2521–2529

[ZWSS16] ZUCKER, GERHARD ; WENDT, ALEXANDER ; SIAFARA, LYDIA ; SCHAAT, SAMER: A cognitive architecture for building automation. In: *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE* : IEEE, 2016, S. 6919–6924