A Declarative C++ Agent Platform for Agent-based Edge Computing

Fabrizio Messina[1], Corrado Santoro[1] and Federico Fausto Santoro[1]

[1]Dipartimento di Matematica e Informatica, University of Catania, Italy

Abstract

Software agents and multi-agent systems are pivotal in the development of distributed autonomous systems. While various agent platforms have been proposed, only a selected few are considered state-of-the-art. These agents exhibit a spectrum of intelligence, from reactive behaviours to advanced reasoning and planning. Integrating these capabilities within multi-agent systems enables distributed artificial intelligence, a programming paradigm where autonomous entities collaborate intelligently across a network to achieve common objectives. The advent of 5G technology is reshaping distributed systems, shifting computation towards the network's edge. This transition has led to a proliferation of embedded devices with microcontrollers, known for low power consumption and not so ample computing resources. In this context, this paradigm aligns seamlessly with small cooperating entities dispersed throughout the network. However, providing agent platforms for embedded devices is challenging. Current state-of-the-art MAS platforms, mostly Java-based, pose concerns about memory and speed overhead for microcontrollers.

Addressing these challenges, we introduce DEMOCLE, a multi-agent platform for embedded systems. DEMOCLE enables logic/declarative agent programming directly in C++, utilising object orientation, macros, and recent language features like lambdas. The runtime system is lightweight, incurring minimal overhead in both time and space. This paper offers a comprehensive overview of DEMOCLE, showcasing its features, advantages, and architecture. Additionally, it provides a practical application example.

Keywords

multi-agents, edge computing, internet of things, networks

CEUR-WS.org/Vol-3579/paper16.pdf

1. Introduction

Since a long time, software agents and multi-agent systems (MAS) have proven their effectiveness in the development of distributed autonomous systems. A large variety of agent platforms have been proposed for this reason, even if nowadays only a few of them are considered the state-of-the-art [1].

A key aspect of software agents is their capability to exhibit a certain form of intelligence, ranging from simple reactive behaviours to more complex ones, that include activities like reasoning, planning, pattern identification, etc. Embedding these characteristic inside multi-agent systems enables the so-called *distributed artificial intelligence* a programming paradigm where several autonomous computing entities, spread over the network, cooperate in an intelligent way to achieve a common goal. However, the way in which distributed systems are being designed and implemented nowadays is changing drastically, above all with the introduction of

WOA 2023: 24th Workshop From Objects to Agents, November 6–8, Rome, Italy

messina@dmi.unict.it (F. Messina); santoro@dmi.unict.it (C. Santoro); federico.santoro@unict.it (F. F. Santoro)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

the 5G technology: the trend is to move computations away from big datacenters and server farms towards the "edge" of the network in the view of splitting the complexity into several cooperating pieces running in small computing platforms placed at the edge of the network. This is the reason why the technology market is currently offering a large number of embedded devices equipped with microcontrollers (MCU), such as Arduino, ESP32 or the STM32 NUCLEO system, that feature very low power consumption, adequate computing resources and, above all, one or more types of wireless connectivity systems (WiFi, LoRA, Bluetooth, BLE, etc.). Together with hardware devices, there is also a wide number of software platforms and libraries that allow developers to write software for such computing devices.

In this new view of distributed computing, the MAS paradigm perfectly fits the concept of having small cooperating computing entities spread over the network, but the main problem is the availability of agent platforms for embedded devices. Indeed, the state-of-the-art of MAS platforms is mainly featured by Java-based tools, such as JADE [2], and even if there exist Java runtimes also for embedded systems, their use is not recommended due to the unavoidable memory and speed overhead that is very critical for a MCU that usually features few megabytes of program and data memory, and clock speeds in the order of hundreds of megahertz. Moreover, if we think about intelligent features, while in the area of deep learning some tools and libraries are now available to run neural networks inside a MCU, the same does not apply for other kind of AI paradigm, such as logic/symbolic reason, BDI, etc.

With these concepts in mind and on the basis of the experience of our research group in the context of symbolic reasoning, we developed Democle, a multi-agent platform for embedded system able to offer a new way to program logic/declarative agents directly in C++ using a BDI paradigm. This aim is reached by strongly exploiting object orientation, macros, together with some features of recent versions of the language, such as lambdas. The runtime system is particularly light and does not add a sensible overhead both in time and space.

The paper describes Democle, showing its features, advantages and architecture, and completing the description also with an example of its usage in a real application.

The paper is structured as follows. Section 2 deals with related work. Section 3 describes the Democle platform providing all of its features. Section 4 presents a case-study, thus showing the effectiveness of Democle. Section 5 reports our conclusions.

2. Related Work

The survey [1] offers a deep dive into the integration of multi-agent systems in edge computing. It meticulously examines various facets, including system architecture, resource management, task allocation, and optimisation techniques. The paper provides a road map for researchers and practitioners to harness multi-agent systems in the dynamic landscape of edge computing.

We already cited JADE [2] as (one of) the most widely used agent platform. While being extensively adopted, the fact that it is written in Java does not make it suitable for MCU-based platforms. Even if there exists Java runtimes for MCUs [3], they feature an unavoidable overhead in terms of execution speed and flash memory occupancy.

Similarly, there are many Python-based tools, such as SPADE, PROFETA [4] and PHIDIAS [5]. Even if Python runtimes are available for MCUs, such as microPython, often such runtimes do

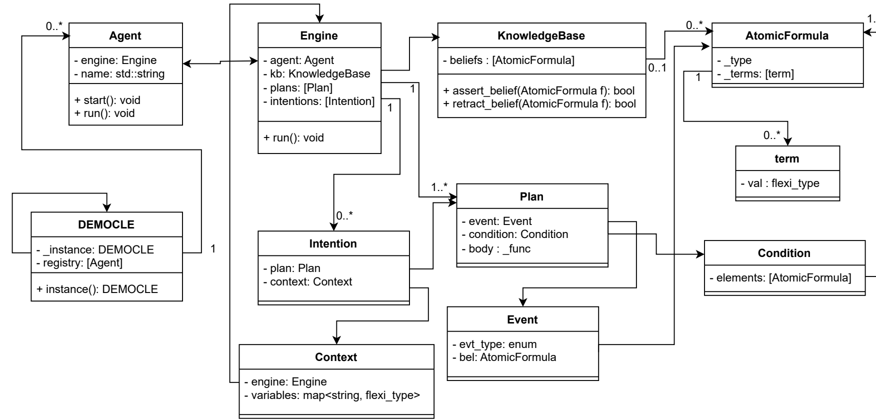

Figure 1: Architecture of DEMOCLE

not offer all the features of the language thus forcing the developer to patch the source code of the platform.

As for C++ based platforms, we can cite Mobile-C [6], BESA-ME [7], EmSBOT [8] and ObjectAgent [9]. While the first is FIPA-compliant, the other provide their own architecture, agent abstraction, and ways to program agent's behaviour. Some of them are also able to feature real-time characteristics since are able to run upon executives such as FreeRTOS. While all of them provide an interesting solution for the MCU world, none of them allows a developer to write declarative plans using the BDI paradigm, which may be useful when a form of (symbolic) reasoning is required in the application.

In the author's knowledge, the only proposal for BDI programming for MCUs is Embedded-BDI [10], which is a development platform to write agents with the AgentSpeak [11] language. An ad-hoc toolchain is provided that uses a pre-compiler, derived from the Jason platform [12], to convert the AgentSpeak program into C++ headers and then compiles everything (AgentSpeak program + runtime) in a single binary image to be uploaded on the target MCU. While the solution is interesting, the programming language is not complete and allows only the specification of the plans in terms of belief matching and manipulation. The interface with the peripherals of the MCU, to generate actions for the external world or to sense the environment, must be written in C/C++ and then integrated at compile time, thus forcing the developer to deal with two different languages and environments. In this sense, the main advantage of DEMOCLE is that the programmer uses only C++ but, thanks to operator overloading and the macros provided by the runtime, is able to express, in the same C++ code, proper statement to define plans in the BDI style.

3. Architecture and Features of DEMOCLE

DEMOCLE is a multi-agent platform to develop MASs with embedded devices exploiting the BDI paradigm. In this sense, DEMOCLE basically uses a logic/declarative approach by offering a series of constructs to represent beliefs, to manipulate and query the knowledge base, and to

express reactive and proactive plans. This is achieved by providing *(i)* a platform to define and implement agents together with their behaviour, and *(ii)* an object library and a macro package to express agent's behaviour by means of a declarative language. DEMOCLE can be considered the successor of PROFETA [4] and PHIDIAS [5], two similar agent platforms developed by the same research group that allows the implementation of BDI agents using the Python language; however DEMOCLE provides a tighter connection between the imperative and declarative part, and it can run without any problems onto MCU-based hardware.

In the following subsection we will describe first the software architecture of DEMOCLE and subsequently the characteristics of the language provided.

3.1. DEMOCLE Architecture

DEMOCLE is a platform organised in an object-oriented way strongly exploiting the features of the C++ language. Its object-based architecture is shown in Figure 1.

The basic entity is the **`Agent`** class which represents a single agent. Developers can subclass it to create the agents of the application and implement the behaviour by means of the specification of one or more *plans* using the declarative syntax. An agent is characterised by a literal *name*, expressed in the class' constructor; all created agents are gathered in the singleton instance of the **`DEMOCLE`** class. Each agent, when started, runs in a thread so it's up to the underlying executive (or operating system) the way in which concurrent execution and scheduling are performed.

An object of the **`Engine`** class is associated to each agent; this object has the task of concretely executing the plans. It has the complete knowledge of the agent's defined plans and, by means of an infinite loop, continuously selects the appropriate plan to be executed according to the BDI paradigm. The Engine is also tied to the **`KnowledgeBase`** which acts as the repository of the beliefs of the agent and also contains the proper code to perform belief insertion/removal, belief query and belief unification.

A *belief* is represented as an instance of the **`AtomicFormula`** class; indeed this class is used for many others entities of DEMOCLE, as it will be detailed in the paper below, in these cases the attribute `_type` represent the kind of entity the object represent. Since, according to traditional declarative/logic programming, an atomic formula can have one or more paramters, that in turn can be ground terms or variables, the `AtomicFormula` class is designed to receive, in the constructor, a variadic number of parameters that are stored in the attribute `_terms` that is a vector of **`term`** objects. Indeed the `term` class is a wrapper that represent either a *constant value* or a *variable* that can be free or bound.

As it has been said above, the `Engine` object tied to an agent contains the set of the plans of the agent, represented as a vector of **`Plan`** objects. Conceptually, a DEMOCLE *plan* is defined according to the same principles of similar frameworks, i.e. PHIDIAS, PROFETA and AgentSpeak, and is featured by:

- a *plan head*, that can be either an event related to the modification of the knowledge base (adding or removing a belief), or a procedure call;
- a *condition*, that is a PROLOG-style predicate that performs a query on the knowledge base;

- a *plan body*, that is a C++ code that is executed when the plan head matches the event/call and the related condition is true.

According to the BDI model, when an event occurs all plans whose head and condition match the event become *intentions*, i.e. instances of the **`Intention`** class. The difference of an Intention w.r.t. a Plan lies in the fact that the free variables of a plan become bound to the proper variable according to the event occurred and the result of the query to the knowledge base, if present; variables and associated values are stored in a **`Context`** object, therefore an Intention is mainly a container of the Plan activated and the Context that represents bound variables. When an Intention is selected for the execution, the associated plan's body is called and the Context object is passed as parameter, so that the C++ code can receive variable values and act properly.

3.2. Overview of DEMOCLE Plan Syntax

Writing plans in DEMOCLE implies to use a proper syntax that is C++ in its nature, but, since it uses some operators, the meaning the statements must be thought in a logic/declarative way. The syntax is derived from that of PROFETA and PHIDIAS, which, in turn, is inspired by AgentSpeak. Plans uses *beliefs* and *variables*, that, before appearing in the code, must be declared. *Beliefs* are declared as global symbols using the macro **`belief(`*`bel_name`*`)`**, while *variables* are declared as agent-local symbols by means of the macro **`var(`*`var_name`*`)`**. Plans are then defined in the body of the method **`run()`** of the subclass of the Agent class. We refer to Figures 2 and 3 to explain the way in which the plans of an agent can be expressed. The code shows the implementation of the case study described in Section 4; in the following we will deal with the syntax, while a comprehensive explanation of the functioning of the code is reported in the relevant section.

A *reactive plan* is defined by using the **+** or **-** symbol followed by the specification of a belief with zero or more ground terms or free variables. These plans are triggered by the occurrence of an assertion or retraction of the relevant belief in the knowledge base of the agent.

A *proactive plan*, which is something like a function or procedure, is expressed by using an atomic formula as the head with zero or more ground terms or free variables. These plans are triggered by a specific call from the body of another plan.

Both reactive and proactive plans can be defined with one or more *clauses*, i.e. plans with the same head or triggering event but differing in the *condition* part, which is a predicate made of one or more belief-checks in the knowledge base also using variables to be unified, in the PROLOG style. Condition is defined using the **/** symbol after the plan's head and expressing beliefs in sequences using the connective **&**.

Plan's body appears after the condition following the symbol **»** and is represented by a C++ functor that receives as parameter a Context object; this object can be used to retrieve the values of the variables that are bound by the DEMOCLE engine during the event/condition match.

3.3. Additional Features of DEMOCLE

There are two important additional features that are provided by DEMOCLE, they are *sensors* and *agent communication.*

```
1  #include "democle.h"
2
3  belief(temperature);
4  belief(heartbeat);
5
6  singleton(temp_max);
7  singleton(heartbeat_max);
8
9  reactor(adc);
10 reactor(beat);
11
12 procedure(calc_max);
13
14 class Temperature : public Agent {
15 public:
16     Temperature() : Agent("temperature") {};
17     void run() {
18         var(T); var(H); var(T1); var(H1);
19         AnalogInputSampler * as = new AnalogInputSampler("A0", A0, -55.0, 150.0, 3600);
20         attach(as);
21         +adc("A0", T)  >> [T](Context & c) {
22             float t = c[T];
23             time_t raw_time;
24             struct tm *complete_time;
25             time(&raw_time); complete_time = localtime(&raw_time);
26             c + temperature(t, complete_time.tm_hour);
27             c << "risk_evaluator" << temperature(t, complete_time.tm_hour);
28         };
29
30         +temperature(T,H) >> [T,H](Context & c) {
31             c << calc_max((float)c[T], (int)c[H]);
32         };
33
34         calc_max(T,H) / (temperature(T1,H1) & gt(T1, T)) >> [T1,H1](Context & c) {
35             c << calc_max((float)c[T1], (int)c[H1]);
36         };
37
38         calc_max(T,H) >> [T,H](Context & c) {
39             c + temp_max((float)c[T], (int)c[H]);
40         };
41     };
42 };
43 ...
```

Figure 2: Listing 1 of the case-study

Sensors are objects (subclass of **`Sensor`**) that are used to make agents get data from the external world. A sensor is attached to an agent by using the statement **`attach(...)`** in the agent's `run()` method, passing the Sensor object as parameter. A (subclass of) `Sensor` must implement the **`sense()`** method that must contain the code performing the acquisition of the data and generating, as a result, a belief asserted in the agent's knowledge base so that a plan can be triggered.

Even if the developer can easily design her/his own sensors, DEMOCLE offers a sensor library[1]

[1] In the current version, the library contains few sensors but we plan to expand it during the development of the platform.

mainly designed to interface the most widely used peripherals of a microcontroller, and in particular:

- **Timer**, that generates a **`tick()`** belief[2] with a given periodicity;
- **DigitalInputHandler**, that generates a user-defined belief when a rising or falling edge is detected in a digital input pin;
- **AnalogInputSampler**, that performs periodic sample of an ADC channel, generating the **`adc()`** belief for each sample acquired.

The other important feature of DEMOCLE is *agent communication*. This is performed by asserting (or retracting) a belief in the destination agent's knowledge base. To make this possible, the receiving agent must declare the possibility of "opening" its knowledge base to external beliefs sent by other agent. This is done by using the **`accept(...)`** statement together with the specification of the belief's name and its arity. When a message is accepted, it can be received by the agent and, in turn, can trigger a plan in order to have a specific code reacting to the message. In this case, the Context of the plan also hold the name of the sender so that the plan can also include the code to make a reply, if needed.

4. Case-study

To show the effectiveness of DEMOCLE, we report in this section a simple case-study of a MAS for healthcare patient monitoring. In this application, we consider, as the embedded device, a ESP32 Dev-Kit, equipped with a temperature sensor, an heartbeat sensor and wireless connectivity. The framework used to program the device is PlatformIO with the Arduino libraries.

The aim of the application is to measure periodically temperature and heartbeat and, in case of a possible risk, send a proper alert message. The case-study is quite simple, but it is intended to provide a more complete as possible short overview of the various features of DEMOCLE. The code of the example is provided in the listings of Figures 2 and 3.

We consider three agents, **Temperature** and **Heartbeat** that have the task of sampling the relevant data each hour and computing the maximum value in the 24 hours. The single acquisition received by both agents is also forwarded to the **RiskEvaluator** agent, that checks for the risk and, if it is the case, sends an alert message. Temperature and Heartbeat agents have the same behaviour but differs in the data acquired and the beliefs manipulated, so, to avoid a long description, we limit our study to the first agent only.

As Figure 2 shows, after the includes, a DEMOCLE program requires the declaration of the concepts used, i.e. *beliefs* and *procedures*. As the figure shows, DEMOCLE provides also additional types of beliefs that are *reactors* and *singletons* that behave according to a different semantics: *reactors* are beliefs that do not populate the knowledge base but can only trigger reactive plans, so they can be used when an event occurs but does not need to store a specific knowledge; *singletons* are beliefs that can exist in a single instance inside the knowledge base; asserting a yet existing singleton implies to replace the old one (and trigger the plan, if present).

Then the declaration (and implementation) of the Temperature comes, as a subclass of the main Agent class. The method `run()` reports the complete code of the behaviour of the agent.

[2]Indeed it is a *reactor*, a special kind of belief that is described in Section 4.

```
1  ...
2  class RiskEvaluator : public Agent {
3  public:
4      RiskEvaluator() : Agent("risk_evaluator") {};
5      void run() {
6
7          var(V); var(H); var(T);
8
9          accept(temperature/2);
10         accept(heartbeat/2);
11
12         +temperature(T,H) / heartbeat(V,H) >> [T, V, H](Context & c) {
13             c << evaluate((float)c[T], (float)c[V], (int)c[H]);
14         };
15
16         +heartbeat(V,H) / temperature(T,H) >> [T, V, H](Context & c) {
17             c << evaluate((float)c[T], (float)c[V], (int)c[H]);
18         };
19
20         evaluate(T, V, H) >> [T, V, H](Context & c) {
21             float t = (float)c[T];
22             float v = (float)c[V];
23             int h = (int)c[H];
24             if (critical(t, v, h)) send_alert(t, v, h);
25             c - temperature(t,h);
26             c - heartbeat(t,h);
27        };
28
29     };
30 };
31
32 void setup()
33 {
34     Temperature * t = new Temperature();
35     Heartbeat * h = new Heartbeat();
36     RiskEvaluator * r = new RiskEvaluator();
37
38     t->start();
39     h->start();
40     r->start();
41
42 }
```

Figure 3: Listing 2 of the case-study

In the first part (lines 18 and 19), variables used in plans are declared as well as the sensor, the AnalogInputSampler in our case, that is attached to the agent with the statement in line 20.

Lines 21–28 reports the reactive plan that is triggered when the `adc()` belief (reactor) is asserted: indeed this operation is performed by the AnalogInputSampler each hour. As a result, variable T is bound to the temperature value sampled and the plan's body gets the current time and asserts the belief `temperature()` with the value as first parameter and the hour as the second parameter (line 26). The same belief is sent to the RiskEvaluator agent (addressed by name) for further processing (line 27). The last operations are performed by involving the Context object obtained by the plan's body: it is not only the repository of the bound variables but also acts as a proxy of the agent, thus, by using the operators **+**, **-** and **«**, we can add or

remove beliefs, call a procedure or send a message to another agent, as Listing 1 and 2 shows in lines 26, 27, 31, 35 and 39, for Listing 1, and 13, 17, 25 and 26 for Listing 2.

After the assertion of belief `temperature()` the plan defined in lines 30–32 is triggered, which has the aim of computing the maximum temperature of the day till now. Such an operation is performed by the plans related to procedure `calc_max()` reported in lines 34–40. As the reader can see, there are two plans (two clauses) having both the same head, but the former includes a condition part. In detail, plan in lines 34–36 is triggered when another belief `temperature()` is found in the knowledge base but with the value of T1 greater than[3] the current maximum temperature (variable T); as a result the new temperature becomes the new temporary maximum and the procedure is called recursively. But when the condition is false, we have found the absolute maximum so the plan in lines 38–40 can be triggered that, as the action, stores the maximum temperature (and the hour in which it has been sampled) in the knowledge base using the proper belief.

Agent Heartbeat features the same behaviour of Temperature, but uses a different sensor and the beliefs `heartbeat()` and `heartbeat_max()` to store the current and the maximum value of the heartbeat sensed.

The last agent of the application is RiskEvaluator, whose code is reported in Figure 3. After class definition and variable definition, the listing shows the presence of the `accept()` statement, which expresses the fact that the specified beliefs, also with the given number of parameters (arity), can be accepted by this agent as external messages sent to it. Since this agent has the aim of processing both temperature and heartbeat data, two reactive plans are defined: the former triggered by the `temperature()` belief, the latter triggered by the `heartbeat()` belief. Both plans checks for the presence of the other belief in the knowledge base, but given that the hour parameter is the same (otherwise correlation is not possible), and, as a result, call the `evaluate()` procedure (lines 20–27): it checks if an alert is worth to be sent and then (lines 25 and 26) removes the received beliefs from the knowledge base because they are no more useful.

The listing in Figure 3 ends with the C++ `setup()` function which is the one executed by the Arduino framework when the program is started. It shows how agents are created in DEMOCLE: they must be simply instantiated and then started by calling the method `start()`.

5. Conclusions

This paper has introduced DEMOCLE, C++ a platform that allows developers to program MASs using the BDI paradigm for MCU-based hardware and thus suitable to build MASs for IoT/Edge computing. The peculiar feature of DEMOCLE is the possibility of specifying beliefs and plans in logic/declarative manner, embedding the declarative code inside the same C++ source code. This is made possible thanks to the use of operator overloading and lambda functions, two very important characteristics provided by the C++.

DEMOCLE is currently in an early development stage even if the logic/declarative engine is complete. New features are planned, such as a simpler way to access Context variables, additional

[3] The statement `gt(T1,T)` is a library predicate and returns true when the first parameter is greater than the second one.

sensor classes for all the peripherals of MCUs and a flexible way to perform distributed agent communication, among different devices and using different wireless communication protocol.

Democle is released with the BSD license and can be obtained through github at the URL http://github.com/corradosantoro/democle.

Acknowledgements

This work is supported by the MIUR PRIN project "T-LADIES".

References

[1] A. Dorri, S. S. Kanhere, R. Jurdak, Multi-agent systems: A survey, Ieee Access 6 (2018) 28573–28593.

[2] F. L. Bellifemine, G. Caire, D. Greenwood, Developing Multi-Agent Systems with JADE, Wiley, 2007.

[3] Stm32 f4 series java evaluation kit - with stm32f407ig mcu, 2023. Https://www.st.com/en/evaluation-tools/stm3240g-java.html.

[4] L. Fichera, F. Messina, G. Pappalardo, C. Santoro, A Python Framework for Programming Autonomous Robots Using a Declarative Approach, Sci. Comput. Program. 139 (2017) 36–55. URL: https://doi.org/10.1016/j.scico.2017.01.003. doi:10.1016/j.scico.2017.01.003.

[5] Phidias web page, 2019. Https://github.com/corradosantoro/phidias.

[6] B. Chen, H. H. Cheng, J. Palen, Mobile-c: a mobile agent platform for mobile c/c++ agents, Software: Practice and Experience 36 (2006) 1711–1733. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.742. doi:https://doi.org/10.1002/spe.742. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.742.

[7] Y. Díaz, M. Flórez, E. González, Multiagent systems for embedded applications: Besa mobile embedded, in: 2012 7th Colombian Computing Congress (CCC), 2012, pp. 1–6. doi:10.1109/ColombianCC.2012.6398035.

[8] L. Peng, F. Guan, L. Perneel, M. Timmerman, Emsbot: A modular framework supporting the development of swarm robotics applications, International Journal of Advanced Robotic Systems 13 (2016) 1729881416663662. URL: https://doi.org/10.1177/1729881416663662. doi:10.1177/1729881416663662. arXiv:https://doi.org/10.1177/1729881416663662.

[9] D. Surka, M. Brito, C. Harvey, The real-time objectagent software architecture for distributed satellite systems, in: 2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542), volume 6, 2001, pp. 2731–2741 vol.6. doi:10.1109/AERO.2001.931294.

[10] Embedded-bdi, 2023. Https://embedded-bdi.github.io/.

[11] A. Rao, AgentSpeak (L): BDI agents speak out in a logical computable language, Lecture Notes in Computer Science 1038 (1996) 42–55.

[12] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology), John Wiley & Sons, Inc., Hoboken, NJ, USA, 2007.