P2P electric prosumers and MAS

Lorenzo Foschi

lorenzo.foschi.work@gmail.com

05/02/2025

1. Details of the proposal

1.1 Full specification of your proposal

The purpose of the proposal is integrating MAS knowledge into the existing scholarship research project on my energy-prosumers network tool.

Aiming towards a real BDI approach in that context includes three different directions, both theoretical and practical:

1) A theoretical presentation, with a chance of having code mockups on specific features, on the extensibility of prosumer agents design modelling towards real BDI architecture. This involves iterative steps, starting from environmental definition, passing through neighbours awareness and inner prosumer definition, up to communication protocols among them: Section **3.**

2) An implementation of BDI prosumers as conceived in point 1, with suitable simplifications that can make the work fit the 7-10 days required by the project, in the python-agentspeak framework (https://github.com/niklasf/python-agentspeak) and a comparison of how they would have been implemented in Jason (at least in theory, but if possible practical, with the same pieces of code implemented in both environments): Section **4.**

3) Brief presentation and integration of the "mango" framework (1*) with my research tool (2*), to show how non-BDI agents implemented in the mango simulator could be seamlessly integrated in an existing simulator for the prosumer model: Section **5.**

While points 1 and 2 will be surely achieved (with due simplifications and assumptions in point 2), point 3 will be addressed only if there is time left.

(1*)

Mango (https://www.sciencedirect.com/science/article/pii/S2352711024001626) is a multi-agent python tool already coupled with the "cosima" (https://github.com/OFFIS-DAI/cosima) library that we use for the research project.

Both frameworks come from OFFIS so they're ready to be coupled together following some documentation (https://cosima.readthedocs.io/en/latest/Coupling_with_mango.html). However, this doesn't mean that simulators inside the environment will be able to work under such new setup.

I already had a conversation and a call with main researchers from OFFIS in Oldenburg:

M.Sc. Malin Radtke

Wissenschaftliche Mitarbeiterin | Researcher

OFFIS e.V. - Institut für Informatik

FuE Bereich Energie | R&D Division Energy

Escherweg 2, 26121 Oldenburg - Germany

E-Mail: malin.radtke@offis.de

URL: www.offis.de

Registergericht: Amtsgericht Oldenburg VR 1956

Vorstand: Prof. Dr. Sebastian Lehnhoff (Vorsitzender), Prof. Dr. techn. Susanne Boll-Westermann, Prof. Dr.-Ing. Andreas Hein, Prof. Dr.-Ing. Astrid Nieße

(2*)

As reference, a short abstract of the prosumer project is presented in the **2.1.1** section; where "Mosaik" [5] is a co-simulation framework (which allows to run multiple simulators in a shared environment), "OMNeT++" [6] is a network simulator (which allows to exchange inet messages) and "Cosima" [4] is the py project that couples these two frameworks together.

Link of Github repo is: https://github.com/ihcsof/STONKSpp

STONKSPP

SOFTWARE TOOL FOR ORCHESTRATING NETWORKS: A KIT OF SIMULATORS OF P2P PROSUMERS

1.2 The kind of your proposal

Creative

1.3 The range of points/difficulty of your proposal

Hard

> *"It requires pretty deep understanding on both course concepts and underlying research topics covered in my paper"*

Note: LLM was leveraged for minor polishing of author-written text; in portions with particularly articulated English content, where it could enhance readability by bridging the language gap.

2. Introduction

2.1 Starting point is my paper: STONKSpp

2.1.1 Abstract

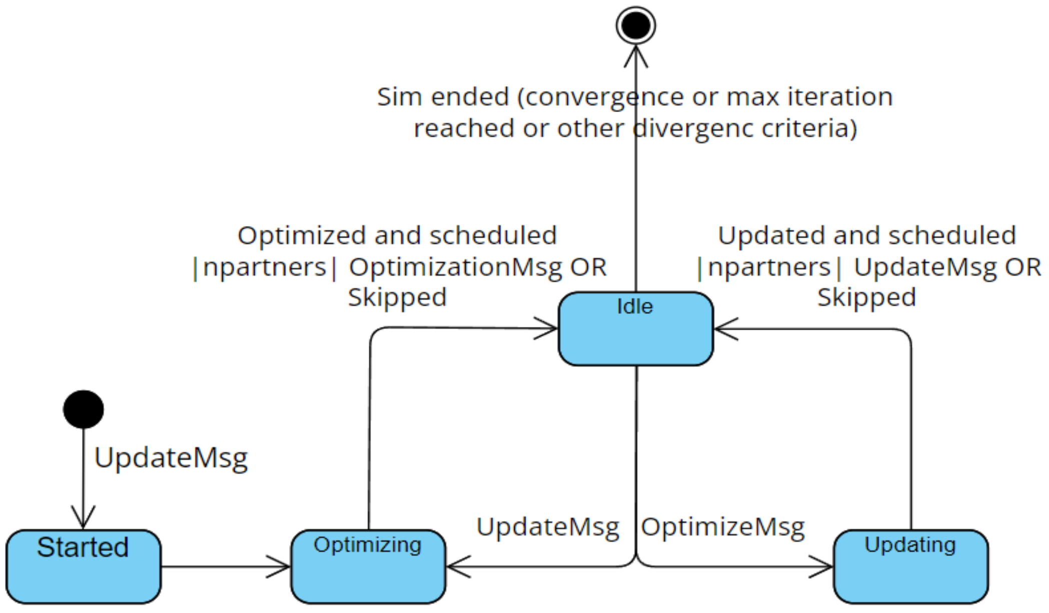
The paper “*Software Tool for Orchestrating Networks: a Kit of Simulators of P2P Prosumers (L.Foschi, M.Dell’Amico, G.Ferro, G.Longo, E.Russo) [1]”* presents the development and implementation of STONKSpp, a comprehensive toolkit designed for simulating Peer-to-Peer (P2P) prosumer networks in energy markets. The toolkit enhances existing models by integrating network simulations, thereby addressing the limitations of previous approaches that assumed ideal network conditions. Key features include transitioning from a sequential to a message-passing paradigm, removing graphical user interfaces to optimize performance, and integrating multiple simulators using the Mosaik and OMNeT++ frameworks. The results demonstrate significant improvements in simulation realism, particularly in accounting for network latencies, disruptions, and data loss. This toolkit provides a valuable resource for researchers and practitioners in the field of smart grid simulations, enabling more accurate and applicable real-world scenario testing.

2.1.2 Implementation details

In the initial code (from “Baroche’s Prosumer markets: A unified formulation [2]”) the network was assumed to be perfect (no message loss) and perfectly synchronous: at every iteration, each node received all messages sent to it in the previous timeslot, processed the messages, and sent updates to neighbors. This was implemented straightforwardly with a few nested loops, where an optimize method was called for each prosumer.

A more detailed network simulation required a drastically different implementation, and this was the main focus of our research work.
This involved transitioning to a message-passing paradigm where receiving messages triggered changes in node statuses (e.g., optimizing, updating, idle). This was achieved using a discrete-event simulation technique, where simulated events were placed in a priority queue, prioritized by the simulated time at which the event occurred. Processing events could trigger other events that were added to the queue (e.g., receiving an update from other nodes would put a node in the optimizing state and schedule a new event for when optimization was complete).

The main optimization function was initially invoked within a primary while loop. Nested within that was another while loop containing a for loop, executed once per iteration. Within this, an optimize method was called for each of the prosumers. This sequential structure posed complications for further integrations. To address this, a new solution was devised where each prosumer only exchanged messages with its direct neighbors, sending and receiving only the strictly necessary information. This shift allowed the system to better mimic the distributed nature of real-world energy systems, where prosumers operate independently but must communicate efficiently to maintain system stability.


Sim ended (convergence or max iteration reached or other divergenc criteria)
Optimized and scheduled |npartners| OptimizationMsg OR Skipped
Updated and scheduled |npartners| UpdateMsg OR Skipped
Idle
UpdateMsg
Started
Optimizing
UpdateMsg
OptimizeMsg
Updating

Further on in this report we'll strictly refer to this paradigm change for our purposes.

2.2 Reasoning under this starting point

Once understood the general key-points of the research topic and its underlying developed tool, it's relevant to pinpoint the reasoning under the decision of bringing MAS knowledge in this context:

- Prosumer's smart grids are one of the most prominent multi-agent-related fields. As cited in *CETINA's seminars on Research in Artificial Intelligence for the Energy and Infrastructure Sector (held by S&DAI prof. V.Mascardi)*, intelligent agents have demonstrated their value in peer-to-peer energy trading and managing dynamic scenarios, highlighting their ability to adapt to rapidly changing environments. MAS are characterized by their autonomy, reactivity, proactivity, and social capabilities. Hence, these systems are particularly effective in solving distributed problems, as they operate in environments where information and control are decentralized.

- As seen in Section 1, it's now clear that in the STONKS paper I already implicitly made some steps towards MAS, moving to a decentralized approach with real-message passing among multiple independent entities.

- Nevertheless, further steps can be thought of.
  Due to the inherent usage of Python, without any MAS framework or language, there's indeed no strong multi-agent implementation: this project finds its rationale and motivations here.

3. Extensibility towards BDI approach

3.1 Review of the literature

MAS frameworks have found success in various energy applications, such as optimizing local energy trading, balancing supply/demand and enhancing the efficiency of microgrids.

3.1.1 How Agents Discover Each Other

One of the most critical aspects of MAS in P2P energy markets is how agents find and connect together; typically achieved through decentralized algorithms. Distributed Constraint Optimization Problems (DCOPs *(*1)*) are a popular method, allowing agents to collaborate and optimize resource allocation without needing a central authority. Techniques like branch-and-bound searches *(*2)* make these processes more efficient, even in dense networks where many agents are competing for resources. Communication infrastructures, including IoT devices, play a crucial role here, providing the real-time data exchange necessary for agents to dynamically infer and interact with their peers. Furthermore, blockchain can be leveraged as a new and different way to integrate MAS, *as deeply explained in the Decentralized Systems project*.

3.1.2 The Role of BDI in Agent Behavior

The Belief-Desire-Intention (BDI) model is widely used to design the behavior of agents in these systems. BDI essentially mimics human decision-making by enabling agents to act based on their beliefs (like the current state of the grid), desires (such as reducing energy costs), and intentions (specific actions like negotiating a trade or conserving power). This makes the agents more autonomous and adaptable, particularly in unpredictable environments like a smart grid with fluctuating energy demands and prices. However, little work has been done to properly formalize a model for strict BDI prosumer-agents.

3.1.3 Challenges and Opportunities

Despite its promise, MAS in energy markets faces some challenges. Scalability is a big one, because coordinating many agents and evaluating their interactions can become computationally expensive. There's also the issue of inclusivity, due to vulnerable consumers overlook (e.g low-income households).

3.1.4 Our starting point

The starting point for the transition toward a BDI architecture leverages the STONKSpp code, in which each prosumer is already somewhat autonomous, reacts to messages and pro-actively attempts to optimize its decisions in a distributed manner. However, these behaviours are encoded procedurally and not explicitly separated into (B)eliefs, (D)esires, and (I)ntentions.

*(*1) DCOPs are optimization problems where multiple agents cooperate to find the best assignment of variables, subject to constraints, while minimizing a global cost function.*

*(*2) Branch-and-bound is a search algorithm that systematically explores solution spaces by dividing them into smaller subproblems (branching) and pruning suboptimal solutions using bounds.*

3.2 Environmental definition

3.2.1 Enviroment definition in the context

In a BDI-based system, the environment typically represents both the physical world and the simulation platform that agents perceive. In our case:

- The *physical world* includes the energy network itself and any external constraints (e.g. physical boundaries, weather conditions, market constraints).

- The *simulation platform* includes the discrete-event simulator, message queues, and scheduling policies.

Since our simulation already has a notion of an environment (namely the Mosaik world, which embodies the whole set of agents over the network), the first step is to make this environment visible as a structured set of beliefs within each agent.

3.2.2 Proposals

Given that the environment is currently scattered across multiple entities we should first identify these and consequently study an ideal extraction of the environment.
In particular, pointing out that for each graph structure we can have multiple networks:

- Graph logic is embodied in the Simulator, statically.
- Network logic is embodied in the .ned files and accessed by the World.
- Neighbor set is then extracted and kept in simple array form, as we could be able to freely infer the Prosumer's neighborhood, which from its point of view should be a real task to tackle.
- Cosima's Communication Simulator handles message passing among the entities.

Therefore, we can introduce an abstract BDIenv with the following capabilities:

- Storing global states (e.g. network topology, global time, and external constraints).
- Providing environment-related services for agents (e.g., query the neighbor set, get network latencies, or check resource constraints).

3.2.3 Simple code mockups

Here follow some simplified code mockups, which can help to understand the concept:

```python
class BDIenv:
    def __init__(self, graph, external_data=None):
        self.graph = graph
        self.external_data = external_data or {}
        self.global_time = 0

    def update_time(self, t):
        self.global_time = t

    def get_neighbors(self, agent_id):
        return self.graph.neighbors(agent_id)

    def get_external_signal(self, signal_name):
        return self.external_data.get(signal_name, None)
```

```python
class BDIprosumer:
    def __init__(self, environment_bdi, agent_id):
        self.beliefs = {}
        self.desires = []
        self.intentions = []
        self.agent_id = agent_id
        self.env = environment_bdi
        self.update_beliefs()

    def update_beliefs(self):
        self.beliefs['time'] = self.env.global_time
        self.beliefs['neighbors'] = self.env.get_neighbors(self.agent_id)
```

3.3 Neighbours Awareness

Currently, neighbors are simply stored in the adjacency matrix/list. Therefore, each Prosumer obtains an array of partners from part[x.index], which enumerates its neighbors in the network. In a BDI this translates to:

Belief: "I believe that agents A,B,C are my neighbors". *Desire*: "I desire to maintain a beneficial energy exchange relationship with them". *Goal, and its Intention through instantiation*: "I intend to send messages or propose trades to these neighbors, in order to meet my desire and benefit from that".

In a BDI setup, an agent's knowledge about its neighbors can definitely be dynamic, as it may discover new neighbors or lose existing ones due to changing environmental conditions or reconfiguration events (this already happen but it's mostly handled only on network side). To handle such changes, methods should be provided to enable each agent to reason about neighbor dynamics, such as identifying when a neighbor goes offline or when a new neighbor enters the system. Additionally, an optional advanced extension involves incorporating social awareness and trust, allowing agents to track the reliability or trustworthiness of their neighbors by forming beliefs such as *"Neighbor A frequently fails to deliver energy on time"* or *"Neighbor B is a trustworthy manager"*.

Finally, gossiping algorithms can be leveraged to help Prosumers discover each other during the proposed "zero-knowledge" initliazation phase, updating their belief-base via message-passing (e.g tell/untell in Jason).

3.4 Prosumer definition

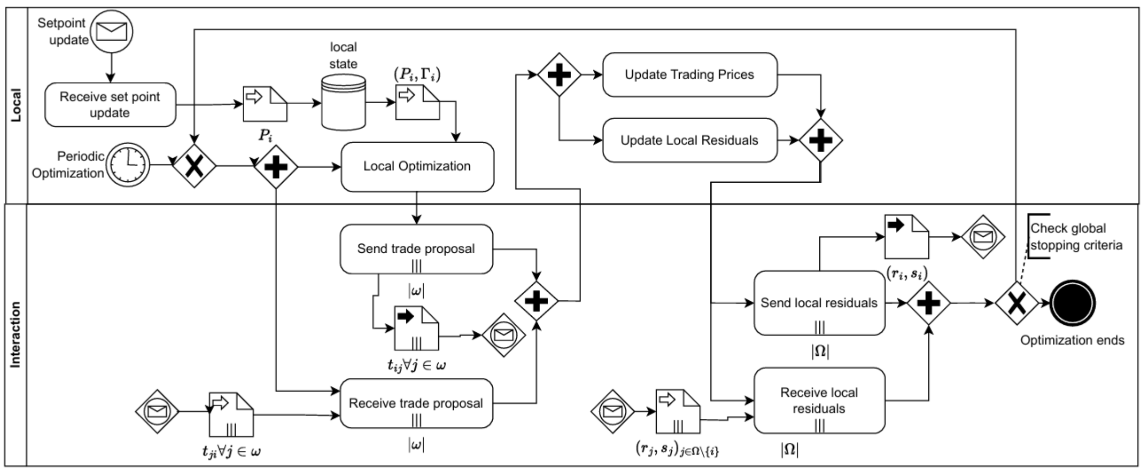
In order to obtain a BDI prosumer definition a mapping table is proposed:

	Currently	*With BDI*
Beliefs	Trades, partner list, a/b cost function parameters, rho penalty factor, estimated electricity demand, observed electricity usage, local/global prices	Set of beliefs that must be true (see context in the pseudocode below)
Desires	Minimizing costs, maximizing social welfare, achieving local constraints (e.g local conspumption <= production + battery), cooperate with neighbors	Body of the plan to execute (see the body of the opt plan in the pseudocode below)
Intentions	Performing local optimization, sending trade proposals to neighbor X with a certain quantity, receiving a trade proposal	Triggering event due to message received & send message to neighbors (see pseudocode below)

```
plan opt:
    trigger: message_received(“UpdateMsg”)
    context: believes(“require_opt”, True)
    body:
        update_local_model()
        run_gurobi()
        update_beliefs_with_new_solution()
        send_message_to_neighbors(“OptimizeMsg”)
```

3.5 Communication Protocols

Before delving deeper into the theoretical mapping of communication protocols, the following BPM diagram by Giacomo Longo is proposed: it includes the entire abstract view of how the AI convergence mechanism is handled with respect to the message paradigm.



3.5.1 Message passing with BDI

Currently the system uses a dictionary-based approach with src;dest;trade, handled by Cosima's CommunicationSimulator to redirect the messages to the right Prosumer's IP addresses. BDI frameworks often rely on speech-act or FIPA-like communicative acts. Instead of just sending raw trades, an agent can send:

Inform: "I (Prosumer A) inform you (Prosumer B) that my new supply is X kW".

Request: "I (Prosumer A) request you (Prosumer B) to send me an updated price".

When agent A believes that it needs more energy, it forms a desire to get it from a neighbor:
Propose: "I (Prosumer A) propose an exchange of Y kW at price P."

Upon receiving *Propose* from neighbor A, prosumer B checks its own beliefs (available resources, cost function, …).

If B's plan to handle the proposal is satisfied, B either sends back accept or reject.
Accept/*Reject*: "I (Prosumer B) accept your proposal" or "I (Prosumer B) reject it."

3.5.2 Simple code mockup

```python
class ProsumerBDI:

    def propose_trade(self, neighbor_id, quantity, price):
        msg = {
            'performative': 'propose',
            'sender': self.agent_id,
            'receiver': neighbor_id,
            'content': {'quantity': quantity, 'price': price}
        }
        self.send(msg)

    def on_message(self, msg):
        performative = msg.get('performative')
        if performative == 'propose':
            self.handle_proposal(msg)
        elif performative == 'accept':
            self.handle_accept(msg)
        elif performative == 'reject':
            self.handle_reject(msg)

    def handle_proposal(self, msg):
        proposed_q = msg['content']['quantity']
        proposed_p = msg['content']['price']

        if self.beliefs['available_power'] >= proposed_q:
            self.send({
              'performative': 'accept',
              'sender': self.agent_id,
              'receiver': msg['sender'],
              'content': msg['content']
            })
        else:
            self.send({
              'performative': 'reject',
              'sender': self.agent_id,
              'receiver': msg['sender'],
              'content': msg['content']
            })
```

4. Basic implementation of the BDI approach

4.1 Python – agentspeak

4.1.1 What is Python- agentspeak?

Python-agentspeak is a Python-based interpreter for the agent-oriented programming language JASON. This library makes it easy to create and manage intelligent agents, offering syntax and functionalities similar to JASON in a Python environment.

4.1.2 Basic implementation

The first snippet will be the python environment, where any py code can be unified with the agents logic. After having imported the library the only actions needed are importing it, defining environment and actions, building the agents from source files and running the environment which embodies them.
Snippets in the following pages were proposed by me as PR in the official Py-agentspeak examples's folder.

```python
#!/usr/bin/env python

import os
import agentspeak.runtime
import agentspeak.stdlib

def main():
    # Create the environment
    env = agentspeak.runtime.Environment()
    actions = agentspeak.stdlib.actions

    # Build the first agent (Prosumer1) from prosumer1.asl
    with open(os.path.join(os.path.dirname(__file__), "prosumer1.asl")) as source:
        agents = env.build_agents(source, 1, actions)
        # 'agents' is now a list containing one agent (Prosumer1)

    # Build the second agent (Prosumer2) from prosumer2.asl
    with open(os.path.join(os.path.dirname(__file__), "prosumer2.asl")) as source:
        agents.append(env.build_agent(source, actions))
        # Now agents has two agents: [Prosumer1, Prosumer2]

    # Run the environment (which runs all agents in agents)
    env.run()

if __name__ == "__main__":
    main()
```

The code below is the first implemented prosumer, which will behave as a Manager.
It starts by defining beliefs such as available power, energy demand and the existence of a neighbor (we'll improve with dynamic discovery in a more advanced version later on in the document). Afterwards, it'll start optimizing the energy by checking if either an energy surplus is present (in that case a trade with the surplus is proposed) or not.
In the first scenario we send an achieve command to the neighbor, proposing a quantity at a fixed price for sake of simplicity.
Then, we define two handlers: one for the trade rejection and one for the acceptance. In the latter case we proceed by proposing another trade, this time with an higher value (which is more likely to get rejected by the neighbor).

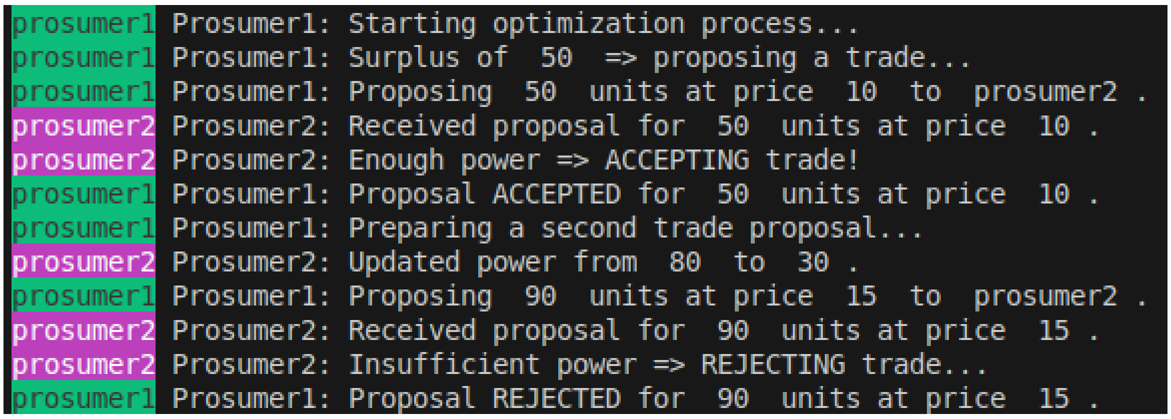
```
/* Beliefs */
available_power(100).
demand(50).
neighbor(prosumer2).

+!start_optimization : true
<- .print("Prosumer1: Starting optimization process...");
   !optimize_energy.

/* Optimize energy (Surplus > 0 => propose a trade) */
+!optimize_energy
:  available_power(Avail) & demand(Dem) & Surplus = (Avail - Dem) & (Surplus > 0)
<- .print("Prosumer1: Surplus of ", Surplus, " => proposing a trade...");
   !propose_trade(prosumer2, Surplus, 10).

/* Optimize energy (Surplus <= 0 => no surplus) */
+!optimize_energy
:  available_power(Avail) & demand(Dem) & Surplus = (Avail - Dem) & (Surplus <= 0)
<- .print("Prosumer1: Not enough surplus (Surplus=", Surplus, "). No trade possible.");
   !no_surplus(Surplus).

/* Propose a trade to another agent */
+!propose_trade(Neighbor, Quantity, Price) : true
<- .print("Prosumer1: Proposing", Quantity, "units at price", Price, "to", Neighbor, ".");
   .send(Neighbor, achieve, propose(Quantity, Price)).

/* If no surplus */
+!no_surplus(Surplus) : true
<- .print("Prosumer1: Surplus=", Surplus, ". No trade possible.").

/* Plans for receiving acceptance/rejection of a trade */
+propose_accept(Quantity, Price) : true
<- .print("Prosumer1: Proposal ACCEPTED for ", Quantity, " units at price ", Price, ".");
   !propose_second_trade.

+propose_reject(Quantity, Price) : true
<- .print("Prosumer1: Proposal REJECTED for ", Quantity, " units at price ", Price, ".").

+!propose_second_trade : true
<- .print("Prosumer1: Preparing a second trade proposal...");
   !propose_trade(prosumer2, 90, 15).

!start_optimization.
```

Then, the following code in the second Prosumer handles the trade in both scenarios.
If the available power in the belief base is enough to met the target trade the acceptance is send via a tell message and the belief base is updated. Otherwise, on the other hand, if the power isn't enough the proposal is simply rejected.

```
/* Beliefs */
available_power(80).

/* If we have enough available_power => accept */
+!propose(Quantity, Price)
:  available_power(Avail) & (Quantity <= Avail)
<- .print("Prosumer2: Received proposal for ", Quantity, " units at price ", Price, ".");
   .print("Prosumer2: Enough power => ACCEPTING trade!");
   .send(prosumer1, tell, propose_accept(Quantity, Price));
   OldAvail = Avail;
   NewAvail = (OldAvail - Quantity);
   -available_power(OldAvail);
   +available_power(NewAvail);
   .print("Prosumer2: Updated power from ", OldAvail, " to ", NewAvail, ".").

/* Not enough power => reject */
```

```
+!propose(Quantity, Price)
:  available_power(Avail) & (Quantity > Avail)
<- .print("Prosumer2: Received proposal for ", Quantity, " units at price ", Price, ".");
   .print("Prosumer2: Insufficient power => REJECTING trade...");
   .send(prosumer1, tell, propose_reject(Quantity, Price)).
```

The following is the resulting output of the execution. If we were to split the prints in multiple parts we would discover that the sequence is not strictly fixed as in Jason's behavior.



```
prosumer1 Prosumer1: Starting optimization process...
prosumer1 Prosumer1: Surplus of  50  => proposing a trade...
prosumer1 Prosumer1: Proposing  50  units at price  10  to  prosumer2 .
prosumer2 Prosumer2: Received proposal for  50  units at price  10 .
prosumer2 Prosumer2: Enough power => ACCEPTING trade!
prosumer1 Prosumer1: Proposal ACCEPTED for  50  units at price  10 .
prosumer1 Prosumer1: Preparing a second trade proposal...
prosumer2 Prosumer2: Updated power from  80  to  30 .
prosumer1 Prosumer1: Proposing  90  units at price  15  to  prosumer2 .
prosumer2 Prosumer2: Received proposal for  90  units at price  15 .
prosumer2 Prosumer2: Insufficient power => REJECTING trade...
prosumer1 Prosumer1: Proposal REJECTED for  90  units at price  15 .
```

4.1.3 Improved implementation

However, in order to met all requirements of the theoretical section, we do need to include the neighborhood awareness. Therefore, in the following code a third agent, namely "neighbors discoverer" is presented: it allows the registration of prosumers by concatenating them to a list, that further on can be retrieved via its getter method.

```
/* Beliefs */
registered([]).

/* Register a new list of prosumers */
+!register(Prosumers) : true
<- registered(Current);
   New = .concat(Current, Prosumers);  /* Concatenate the new prosumers */
   -registered(Current);
   +registered(New);
   .print("DF: Registered prosumers: ", New, ".").

/* Provide the list of registered prosumers */
+!get_prosumers : true
<- registered(Prosumers);  /* Retrieve the list of registered prosumers */
   .print("DF: Sending list of registered prosumers: ", Prosumers, ".");
   .send(prosumer1, tell, prosumer_list(Prosumers)).
```

Finally, in the following snippets we re-define the head of both prosumers:

- In the first prosumer by emptying the neighbors definition in the belief base (then updated after retrieving the newly generated list) and by adding both the registration and discovery logic.

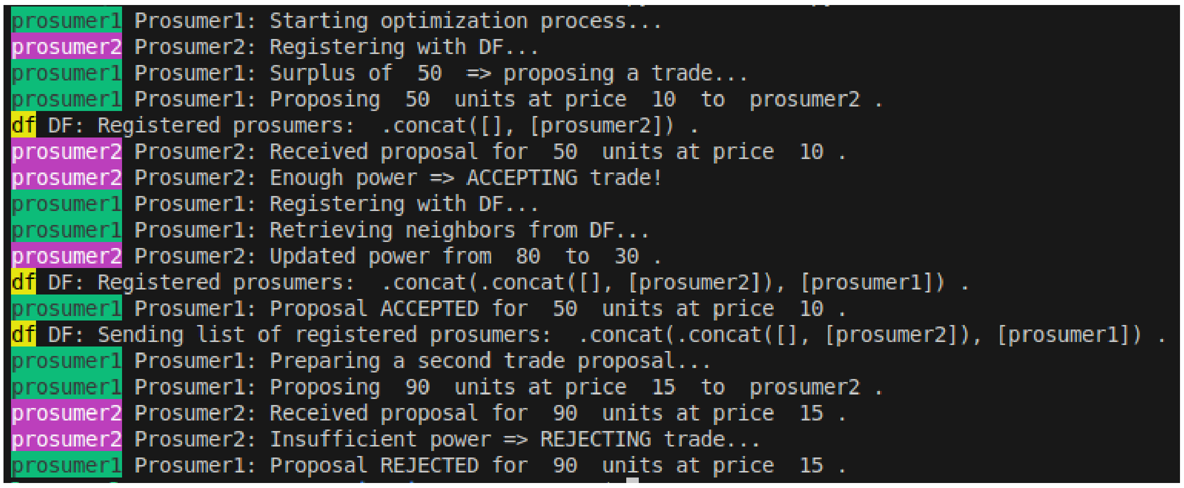
```
/* Beliefs */
available_power(100).
demand(50).
neighbors([]).

```

```
/* Register itself with the DF */
+!start_discovery : true
<- .print("Prosumer1: Registering with DF...");
   .send(df, achieve, register([prosumer1]));
   !retrieve_neighbors.

/* Retrieve neighbors from the DF */
+!retrieve_neighbors : true
<- .print("Prosumer1: Retrieving neighbors from DF...");
   .send(df, achieve, get_prosumers).

+prosumers_list(Prosumers) : true
<- .print("Prosumer1: Received prosumer list: ", Prosumers, ".");
   -neighbors(_);
   +neighbors(Prosumers);
   !start_optimization.
```

- In the second prosumer by just adding the registration logic.

```
/* Beliefs */
available_power(80).

/* Register itself with the DF */
+!start_discovery : true
<- .print("Prosumer2: Registering with DF...");
   .send(df, achieve, register([prosumer2])).
```

The following final image shows how the new execution looks like:



```
prosumer1 Prosumer1: Starting optimization process...
prosumer2 Prosumer2: Registering with DF...
prosumer1 Prosumer1: Surplus of  50  => proposing a trade...
prosumer1 Prosumer1: Proposing  50  units at price  10  to  prosumer2 .
df DF: Registered prosumers:  .concat([], [prosumer2]) .
prosumer2 Prosumer2: Received proposal for  50  units at price  10 .
prosumer2 Prosumer2: Enough power => ACCEPTING trade!
prosumer1 Prosumer1: Registering with DF...
prosumer1 Prosumer1: Retrieving neighbors from DF...
prosumer2 Prosumer2: Updated power from  80  to  30 .
df DF: Registered prosumers:  .concat(.concat([], [prosumer2]), [prosumer1]) .
prosumer1 Prosumer1: Proposal ACCEPTED for  50  units at price  10 .
df DF: Sending list of registered prosumers:  .concat(.concat([], [prosumer2]), [prosumer1]) .
prosumer1 Prosumer1: Preparing a second trade proposal...
prosumer1 Prosumer1: Proposing  90  units at price  15  to  prosumer2 .
prosumer2 Prosumer2: Received proposal for  90  units at price  15 .
prosumer2 Prosumer2: Insufficient power => REJECTING trade...
prosumer1 Prosumer1: Proposal REJECTED for  90  units at price  15 .
```

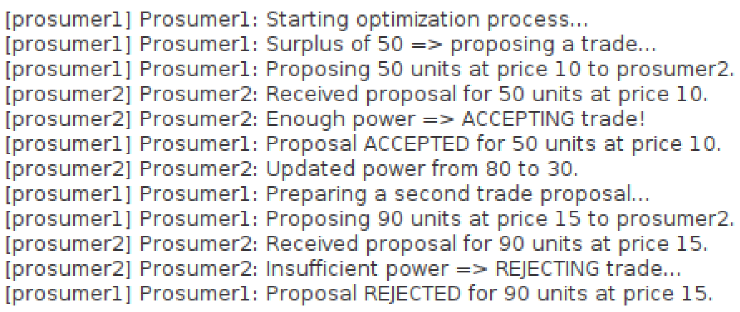
4.2 Comparison with Jason

Deeply inside, Python-agentspeak is a set of lexer/AST/parser classes which allow to use agentspeak syntax in a python environment. Basically, it's like having Jason but with a Python environment behind instead of a Java one.
As reported in the official documentation:

python-agentspeak should be mostly equivalent to Jason.

- *Plan annotations are ignored as of yet.*
- *Standard library does not yet contain syntactic transformations with {begin ...} and {end}.*
- *Standard library does not yet contain introspective and plan-manipulation actions.*
- *Jason 2.0 fork join operators not yet supported.*
- *Literals are only comparable if they have the same signature.*

Therefore, bringing that two .asl Prosumers in Jason brings the same results:

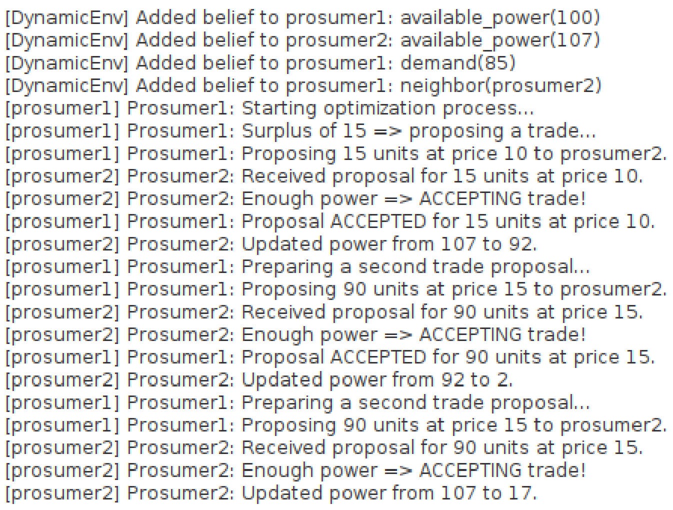


```
[prosumer1] Prosumer1: Starting optimization process...
[prosumer1] Prosumer1: Surplus of 50 => proposing a trade...
[prosumer1] Prosumer1: Proposing 50 units at price 10 to prosumer2.
[prosumer2] Prosumer2: Received proposal for 50 units at price 10.
[prosumer2] Prosumer2: Enough power => ACCEPTING trade!
[prosumer1] Prosumer1: Proposal ACCEPTED for 50 units at price 10.
[prosumer2] Prosumer2: Updated power from 80 to 30.
[prosumer1] Prosumer1: Preparing a second trade proposal...
[prosumer1] Prosumer1: Proposing 90 units at price 15 to prosumer2.
[prosumer2] Prosumer2: Received proposal for 90 units at price 15.
[prosumer2] Prosumer2: Insufficient power => REJECTING trade...
[prosumer1] Prosumer1: Proposal REJECTED for 90 units at price 15.
```

4.3 Dynamic environment

As a last more advanced implementation, a dynamic Jason environment to randomize different runs is presented (variable beliefs, such as power, demand and neighbors):

```java
public void init(String[] args) {
        super.init(args);
        Random rand = new Random();
        String[] agents = {"prosumer1", "prosumer2"};

        for (String agent : agents) {
            int randPower = 50 + rand.nextInt(101);
            String beliefStr = "available_power(" + randPower + ")";
            Literal belief = Literal.parseLiteral(beliefStr);
            addPercept(agent, belief);
            logger.info("Added belief to " + agent + ": " + beliefStr);
        }

        int demand = 50 + rand.nextInt(51);
        String demandStr = "demand(" + demand + ")";
        Literal demandBelief = Literal.parseLiteral(demandStr);
        addPercept("prosumer1", demandBelief);
        logger.info("Added belief to prosumer1: " + demandStr);

        String neighborStr = "neighbor(prosumer2)";
        Literal neighborBelief = Literal.parseLiteral(neighborStr);
        addPercept("prosumer1", neighborBelief);
        logger.info("Added belief to prosumer1: " + neighborStr);
    }
```



```
[DynamicEnv] Added belief to prosumer1: available_power(100)
[DynamicEnv] Added belief to prosumer2: available_power(107)
[DynamicEnv] Added belief to prosumer1: demand(85)
[DynamicEnv] Added belief to prosumer1: neighbor(prosumer2)
[prosumer1] Prosumer1: Starting optimization process...
[prosumer1] Prosumer1: Surplus of 15 => proposing a trade...
[prosumer1] Prosumer1: Proposing 15 units at price 10 to prosumer2.
[prosumer2] Prosumer2: Received proposal for 15 units at price 10.
[prosumer2] Prosumer2: Enough power => ACCEPTING trade!
[prosumer1] Prosumer1: Proposal ACCEPTED for 15 units at price 10.
[prosumer2] Prosumer2: Updated power from 107 to 92.
[prosumer1] Prosumer1: Preparing a second trade proposal...
[prosumer1] Prosumer1: Proposing 90 units at price 15 to prosumer2.
[prosumer2] Prosumer2: Received proposal for 90 units at price 15.
[prosumer2] Prosumer2: Enough power => ACCEPTING trade!
[prosumer1] Prosumer1: Proposal ACCEPTED for 90 units at price 15.
[prosumer2] Prosumer2: Updated power from 92 to 2.
[prosumer1] Prosumer1: Preparing a second trade proposal...
[prosumer1] Prosumer1: Proposing 90 units at price 15 to prosumer2.
[prosumer2] Prosumer2: Received proposal for 90 units at price 15.
[prosumer2] Prosumer2: Enough power => ACCEPTING trade!
[prosumer2] Prosumer2: Updated power from 107 to 17.
```

(Note: one example run)

In python-agentspeak, without having reached a successful implementation due to lack of documentation (*I opened a related Github issue*), the code would adhere to the following schema:

```python
@actions.add(functor="set_belief", arity=2)
def set_belief(agent, term, intention):
    belief_name = str(term.args[0])
    belief_value = int(term.args[1])

    agent.beliefs[belief_name].add((belief_value,))
    yield
```

```python
with open(os.path.join(os.path.dirname(__file__), "prosumer1.asl")) as f:
    prosumer1 = env.build_agent(f, actions)

p1_power = random.randint(50, 120)
p1_demand = random.randint(0, 80)
prosumer1.call(
    Trigger.addition,
    GoalType.achievement,
    Literal("init_beliefs", (p1_power, p1_demand)),
    Intention()
)
prosumer1.call(
    Trigger.addition,
    GoalType.achievement,
    Literal("start_discovery"),
    Intention()
)
```

Then, after having defined a “set_belief” custom action, from the agent code you would call it with the parameters received by the calls in the second part of the snippet (p1_power and p1_demand). The exact same would apply for prosumer2, so here only one side is reported:

```
+!init_beliefs(Power, Demand)
<-  set_belief("available_power", Power);
    set_belief("demand", Demand).
```

5. BDI transitioning on the STONKSpp paper

5.1 Mango Framework

5.1.1 What is Mango?

Mango (modular python agent framework) is a python library for multi-agent systems (MAS), written on top of asyncio. It allows to create simple agents with little effort and in the same time offers options to structure agents with complex behaviour. The main features of mango are: container mechanism to speedup local message exchange, message definition based on the FIPA standard, structuring complex agents with loose coupling and agent roles, built-in codecs such as JSON and protobuf and finally it supports communication between agents directly via TCP or via an external MQTT broker.
As Mosaik and Cosima it was developed by OFFIS-DAI departments, from which I had the pleasure of having some conversations and an entire call, as stated in the proposal text.

5.1.2 Python-agentspeak VS Mango

	PROS	CONS
Python-agentspeak	Simplicity, 1:1 with Jason, strict BDI syntax, FIPA standard	Less powerful, not fully leverage python, no codecs/TCP/MQTT support
Mango	Codecs & TCP/MQTT support, power and features, FIPA standard	Complexity, not directly comparable with agentspeak

5.1.3 Cosima-Mango integration

The cosima-mango project integrates mango into the Mosaik co-simulation with the existing coupling to the communication framework OMNeT++, provided by cosima. By this coupling users can explore how agents interact in a controlled environment, making it an excellent tool for safety-critical domains like the prosumer's one we're describing.

Cosima provides a container simulator that encapsulates a Mango container with agents, allowing these agents to exchange messages via a communication simulator. This setup not only ensures realism in the simulation of distributed systems but also provides a framework to evaluate the robustness and reliability of control mechanisms before deployment.

5.2 Integration trial

5.2.1 Steps to follow

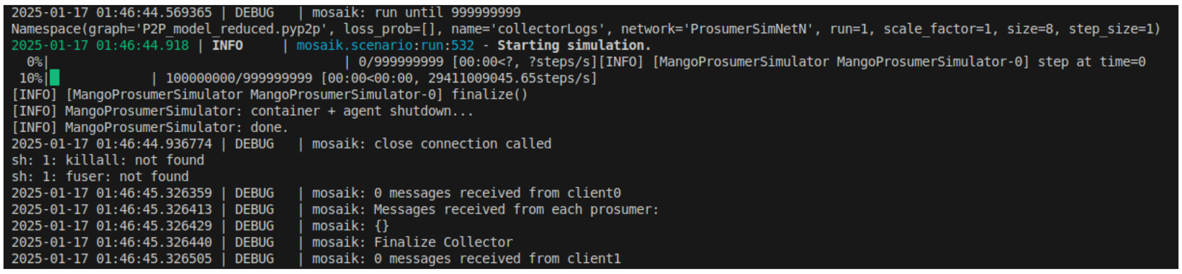
Setting up a simulation scenario involves several well-structured steps. First, users create a scenario script (e.g mango_simulation_scenario.py) that initializes the environment. This involves importing necessary modules, defining configurations, and preparing the simulation environment using Cosima's utilities.
A notable feature of the Mango framework is its role-based architecture, which simplifies defining agent behaviors like ActiveRole and PassiveRole, both inheriting from SimpleRole class, which handles foundational tasks like subscribing to messages, processing them, and managing content exchange.
The ActiveRole builds on this foundation by introducing proactive behavior. It initializes interactions by sending a greeting message to all neighboring agents, therefore initiating the communication cycle, as our Manager of the Python-agentspeak example. On the other hand, the PassiveRole is reactive, responding to messages initiated by active agents.

5.2.2 Basic Mango example

Avoiding to report here code not owned by me, an highly valuable source of mango examples in the context of Cosima are provided by the cosima repository itself.

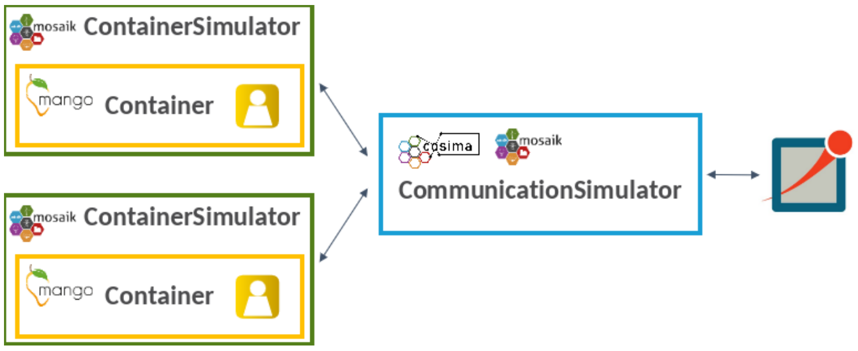
5.2.3 STONKSpp integration trial

At first glance, to implement Mango in the current code, an easy procedure can be followed. However, the problem lies in how Prosumers are implemented. Deeply inside the code, at the bare implementation level, the whole optimization happens at a single node (the original Simulator derived from the *Baroche paper [2]*). Thanks to the new message paradigm it was possible to abstract Prosumer entities as Collectors objects, allowing multiple Ips and real and effective message passing between agents. However, when it comes to real ADMM*(*3)* optimization via (for example) Gurobi, it's always the same "external" Agent that does it. For this reason, once an implementation is tried, zero messages are exchanged between Prosumers due to an architectural misconception:



```
2025-01-17 01:46:44.569365 | DEBUG    | mosaik: run until 999999999
Namespace(graph='P2P_model_reduced.pyp2p', loss_prob=[], name='collectorLogs', network='ProsumerSimNetN', run=1, scale_factor=1, size=8, step_size=1)
2025-01-17 01:46:44.918 | INFO     | mosaik.scenario:run:532 - Starting simulation.
  0%|                                       | 0/999999999 [00:00<?, ?steps/s][INFO] [MangoProsumerSimulator MangoProsumerSimulator-0] step at time=0
 10%|█            | 100000000/999999999 [00:00<00:00, 29411009045.65steps/s]
[INFO] [MangoProsumerSimulator MangoProsumerSimulator-0] finalize()
[INFO] MangoProsumerSimulator: container + agent shutdown...
[INFO] MangoProsumerSimulator: done.
2025-01-17 01:46:44.936774 | DEBUG    | mosaik: close connection called
sh: 1: killall: not found
sh: 1: fuser: not found
2025-01-17 01:46:45.326359 | DEBUG    | mosaik: 0 messages received from client0
2025-01-17 01:46:45.326413 | DEBUG    | mosaik: Messages received from each prosumer:
2025-01-17 01:46:45.326429 | DEBUG    | mosaik: {}
2025-01-17 01:46:45.326440 | DEBUG    | mosaik: Finalize Collector
2025-01-17 01:46:45.326505 | DEBUG    | mosaik: 0 messages received from client1
```

The general procedure implies implementing the code into a new kind of simulator, namely Mango Container, wrapping it into the Mosaik's ContainerSimulator. The latter will exchange messages thanks to Cosima CommunicationSimulator, as it's currently happening with basic Cosima Simulators.



As we can imagine, wrapping the current simulator would require doing something similar to the following code, that will be partially reported due to its length. Nevertheless, this approach can't work in the current setup.

```python
class ADMMRole(Role):
    # other not reported code

    async def _admm_iteration(self):
        if self.has_finished:
            return
        self.iteration += 1

        for i in range(self.nag):
            self.Trades[i, :] = self.players[i].optimize(self.Trades[i, :])
        self.prim = sum(p.Res_primal for p in self.players.values())
        self.dual = sum(p.Res_dual for p in self.players.values())

        log(f'[ADMMRole] iteration={self.iteration}, prim={self.prim}, dual={self.dual}')
        if ((self.prim <= self.residual_primal and self.dual <= self.residual_dual)
                or (self.iteration >= self.maximum_iteration)):
            self.has_finished = True
            return
        self.context.schedule_task(1, self._admm_iteration)

class MangoProsumerSimulator(mosaik_api.Simulator):
    # other not reported code

    def step(self, time, inputs, max_advance):
      # other not reported code
        for msg in output.messages:
            msg_output_time = math.ceil(msg.time * self._conversion_factor)
            out_msg = {
                'msg_id': f'Message_{self._sid}_{self._message_counter}',
                'max_advance': max_advance,
                'sim_time': msg_output_time + 1,
                'sender': self._sid,
                'receiver': msg.receiver,
                'content': msg.message.decode(),
                'creation_time': msg_output_time + 1,
            }
         # other not reported code
```

5.3 Further possible steps

Due to the limited amount of allocated time for the project, which prevents a re-implentation of how agents behaves in the entire simulation, it was impossible to seamlessly integrate Mango. However, the reasoning made for this trial allows to clearly distinguish a step-by-step methodology ranging from the original paper to what it could be a completely decentralized and BDI prosumer network tool:

Baroche, T., Moret, F., Pinson, P., 2019. Prosumer markets: A unified formulation	Completely centralized, monolithic and sequential prosumer exchange code. Not suitable for network simulation.
Towards STONKSpp (1)	Transitioning to message-passing paradigm, but still centralized
Towards STONKSpp (2)	Abstracting prosumers as entities on the network, with 1:1 mapping between exchanged messages from X and proposed trades from the same X
L.Foschi, M.Dell'Amico, G.Ferro, G.Longo, E.Russo. 2025 Software Tool for Orchestrating Networks: a Kit of Simulators of P2P Prosumers	Network simulation over prosumers, with many ways to simulate traffic, delays, algorithms, byzantine behavior and so on.
Towards Mango	Re-implementation of Prosumers as real Cosima simulators, an heavy task!
Actual Mango implementation	Wrapping Mango across Prosumer agents, as done before with the whole Simulator object

*(*3) Alternating Direction Method of Multipliers (ADMM) is an optimization algorithm that breaks problems into smaller subproblems, solving them iteratively and coordinating solutions via dual variable updates to achieve global consensus.*

References

[1] L.Foschi, M.Dell'Amico, G.Ferro, G.Longo, E.Russo. 2025 Software Tool for Orchestrating Networks: a Kit of Simulators of P2P Prosumers – near the review process

[2] Baroche, T., Moret, F., Pinson, P., 2019. Prosumer markets: A unified formulation, in: 2019 IEEE Milan PowerTech, pp. 1–6. doi:10.1109/PTC. 2019.8810474.

[3] Bukar, A.L., Hamza, M.F., Ayub, S., Abobaker, A.K., Modu, B., Mohseni, S., Brent, A.C., Ogbonnaya, C., Mustapha, K., Idakwo, H.O., 2023. Peer-to-peer electricity trading: A systematic review on current developments and perspectives. Renewable Energy Focus 44, 317–333. URL:https://www.sciencedirect.com/science/article/pii/S1755008423000091, doi:10.1016/j.ref.2023.01.008.

[4] Oest, F., Frost, E., Radtke, M., Lehnhoff, S., 2022. Coupling omnet++ and mosaik for integrated co-simulation of ict-reliant smart grids. arXiv preprint arXiv:2209.12550 doi:10.48550/arXiv.2209.12550.

[5] Ofenloch, A., et al., 2022. Mosaik 3.0: Combining time-stepped and discrete event simulation, in: 2022 Open Source Modelling and Simulation of Energy Systems (OSMSES), pp. 1–5. doi:10.1109/OSMSES54027.2022.9769116.

[6] Varga, A., Hornig, R., 2008. An overview of the omnet++ simulation environment, in: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools), p. 60. doi:10.1145/1416222.1416290.

[7] Nair, A.S., Hossen, T., Campion, M. et al. Multi-Agent Systems for Resource Allocation and Scheduling in a Smart Grid. Technol Econ Smart Grids Sustain Energy 3, 15 (2018). https://doi.org/10.1007/s40866-018-0052-y

[8] Deng, Y., An, B. Utility distribution matters: enabling fast belief propagation for multi-agent optimization with dense local utility function. *Auton Agent Multi-Agent Syst* **35**, 24 (2021).https://doi.org/10.1007/s10458-021-09511-z

[9] Inês F.G. Reis, Ivo Gonçalves, Marta A.R. Lopes, Carlos Henggeler Antunes, Towards inclusive community-based energy markets: A multiagent framework

[10] Emilie Frost, Malin Radtke, Marvin Nebel-Wenner, Frauke Oest, Sanja Stark, cosima-mango: Investigating Multi-Agent System robustness through integrated communication simulation, 2024, https://doi.org/10.1016/j.softx.2024.101667.

[11] Viviana Mascardi, Unige, May 3 2024, Seminars on Research in AI for the Energy and Infrastructure Sector (CETINA)

Note: logos are clickable