A Multi-Agent Ontology-based Architecture for Semantic Interoperability and Reasoning in Microgrid Energy Management

Gabriel Santos[1,†], Ricardo Augusto Manfredini[2,*,†]

[1] *University of Trás-os-Montes and Alto Douro, Quinta de Prados 5000-801, Vila Real, Portugal*

[2] *Instituto Federal de Educação, Ciências e Tecnologia do Rio Grande do Sul, Avenida São Vicente, 785, 95174-274, Farroupilha, Rio Grande do Sul, Brasil*

Abstract

Control of distributed microgrids (MG) can be challenging due to the intermittent nature of renewable energy, heterogeneous protocols used in communication, and management. To tackle these restrictions, we propose masIFRS an ontology-based multi-agent architecture that joins edge automation, domain knowledge and predictive models of photovoltaic MGs. We manage five agents within the context of Intelligent Energy Systems Ontology, which includes: Director, Meter, Predictor, Coordinator and Actuator. They communicate using Resource Description Framework messages over Extensible Messaging and Presence Protocol streams. We employ SPARQL templates coupled with a triplestore graph updater for the elevation of real-time data and dynamic updates of transient properties. Additionally, we convert operational policies into executable SPARQL queries via Rules-as-Graphs workflow that allow for live modeling and inference of logic that determines the system when performing energy dispatch storage decisions. To validate this architecture, a simulation was performed with historical electrical and meteorological data obtained from the Federal Institute of Rio Grande do Sul (IFRS), Farroupilha campus. Results show that integration of hierarchical multi-agent coordination and machine learning forecasters improves operational autonomy, reduces optimization cost via caching query results, and minimizes dependence on the central grid given dynamic operating conditions.

Keywords

Microgrid, Multi-agent system, Ontologies, Semantic interoperability, Semantic reasoning

1. Introduction

The rise of Smart Grids has made energy-system control more complex. In microgrids (MGs), we see this complexity clearly: they decentralize generation, their renewable output is intermittent, and their devices operate using different protocols and control strategies. Local automation cannot handle variable supply and demand on its own; therefore, MGs must continually adjust to changing operating conditions. This necessity explains our growing interest in Multi-Agent Systems (MAS), which distribute and decentralize decision-making while maintaining coordination across MGs [1].

MG control depends on specific streams of information: real-time updates, consumption profiles, device statuses, energy-capacity data, and environmental events. To make consistent dispatch, storage, and load-management decisions, the system must read those inputs together. Because moving data alone is insufficient, we consider semantic interoperability essential [2]. Machine learning (ML) now predicts generation and demand with better accuracy, and these advances already transform energy management [3]. However, we observe that researchers still tend to investigate distributed coordination, semantic modeling, and predictive mechanisms separately. Consequently, this lack of integration keeps MGs in a reactive state [4].

Proceedings of FOIS 2026 Satellite events co-located with the 16th International Conference on Formal Ontology in Information Systems (FOIS 2026), September 21-25, 2026, Vitória, Brazil.

[1*] Corresponding author.

[†] These authors contributed equally.

gabsan@utad.pt (G. Santos); ricardo.manfredini@farroupilha.ifrs.edu.br (R. A. Manfredini)

0000-0001-8839-8807 (G. Santos); 0000-0001-5511-9453 (R. A. Manfredini)

Copyright © 2026 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 ISSN 1613-0073

published 2026-11-27

CEUR-WS.org/Vol-4342/paper10.pdf

We ask how we can develop an Energy Management System (EMS) for distributed MGs that integrates operational autonomy, predictive capability, and semantic interoperability through MAS, ontologies, and ML models. We hypothesize that integrating hierarchical MAS, domain ontology, and predictive ML models improves coordination in distributed MGs by ensuring that dispatch and storage decisions reflect actual operating conditions and by reducing reliance on external energy sources. We aim to develop and validate a cooperative energy-management architecture for distributed MGs, named masIFRS, using autonomous agents, semantic models, and ML models. To achieve this goal, we develop a layered ontology for device representation, integrate real-time predictive models for generation and demand, create logical inference mechanisms for dispatch and storage decisions, and evaluate its impact on operating costs and grid dependence.

The foundation of this approach relies on a critical shift in power systems engineering; to facilitate the management of MGs and coordination of distributed energy resources, it is important to shift from centralized architectures like Supervisory Control and Data Acquisition systems [5]. MAS provides robustness and serves as the perfect control infrastructure by modeling electrical components as self-governing software entities that guarantee scalability and fault tolerance [11]. In this respect, modern EMSs in smart condominiums and campuses have employed Hierarchical Distributed Control Mode to structure such complex systems [7, 10], decoupling edge decisions from the cloud and optimizing the communication network by reducing latency at lower costs.

The abundance of non-standard data generated by multiple devices makes interoperability challenging [2], although MAS provides stable control. For this heterogeneous data-consumption profiles, device states and environmental events-layered ontologies hold the electrical or topological structure and commercial factors at a correct granularity level [6]. Semantic models like the Intelligent Energy Systems Ontology (IESO) [2] function as a digital twin. It allows heterogeneous agents to understand operational information and perform real-time validations using SPARQL Protocol and RDF Query Language[2] (SPARQL) [14].

Apart from logic-based semantic reasoning, contemporary EMSs call for proactive resource allocation to alleviate the unpredictability of renewable energy sources and sudden load variations [8]. MAS and ML enhance the predicted future state of the MG. Predictive models and federated deep reinforcement learning protect the privacy of local data [9]. Furthermore, this approach also optimizes battery dispatch to minimize operating costs and decrease dependence on the grid [9]. Despite these developments, fragmentation still limits the domain, as no uniform frameworks yet integrate MAS, ontologies, and ML. We find that this disconnect prevents MGs from becoming proactive. To fill this gap, we propose a cooperative EMS combining operational autonomy, semantic interoperability, and prediction power. These features allow the system to make dispatch and storage decisions based on real operating conditions, while abating dependence on external energy sources.

We describe the architecture and methods in Section 2; we present experiments and results in Section 3; and we conclude and outline future work in Section 4. To establish a comprehensive foundation for understanding the proposed masIFRS model, it is crucial to analyze the underlying concepts of decentralized management, MASs, ontologies, and ML applied to MG optimization.

2. Proposed Framework (Methodology)

The proposed framework comprises a distributed architecture based on interoperable intelligent agents using semantic models for reasoning and message exchange. This section presents the multi-agent architecture, the ontologies (re)used, and how they are leveraged for agent reasoning.

2.1. Agent Architecture

The computational architecture of the masIFRS multi-agent system, shown in Figure 1, is a decentralized and logically interoperable framework for managing MGs, which enables real-time decision-

[2]Recursive acronym.

making. These agents utilize the Smart Python Agent Development Environment (SPADE) because it supports asynchronous agent actions. The agents communicate with each other by sending structured messages via the Extensible Messaging and Presence Protocol (XMPP), which ensures they communicate control data and actuator signals in a decentralized manner [5].

All the persistence of any historical operations and time-series datasets, such as photovoltaic generation, electrical consumption, battery state of charge and ML projection is managed through PostgreSQL. Meanwhile, the GraphDB plays a semantic database role in such scenarios by managing knowledge representation and interoperability. It contains the domain ontology and provides the capabilities needed to perform logical inference as well as validate runtime business rules using SPARQL queries [2]. The use of these four technologies — SPADE, XMPP, PostgreSQL and GraphDB — provides the necessary support for intelligent and coordinated system action. From this technological base, the following subsections describe in detail the operational scope, responsibilities and communicative acts of each of the five agents that make up the masIFRS architecture: Director, Coordinator, Meter, Predictor and Actuator.

The masIFRS life cycle starts when agent Meter runs on the edge layer to harvest data of all MG elements and continuously takes snapshots of current consumption, photovoltaic generation and energy storage system status. At the same time, the Predictor agent works on historical data and uses ML methods to forecast generation and demand stochastic behavior in the short term. The Coordinator agent collects the information from both Meter and Predictor data. The Coordinator checks the new state of the network, in accordance with a set of operating policies that are created and modified on the fly by the Director, who is responsible for building and managing business rules. The Coordinator uses imperative commands via SPARQL query to the Actuator agent whenever the rule condition is met (by inference). This will serve the control loop and ends at the Actuator, which executes these commands as physical actions on the distributed devices and updates their new state in the semantic knowledge graph.

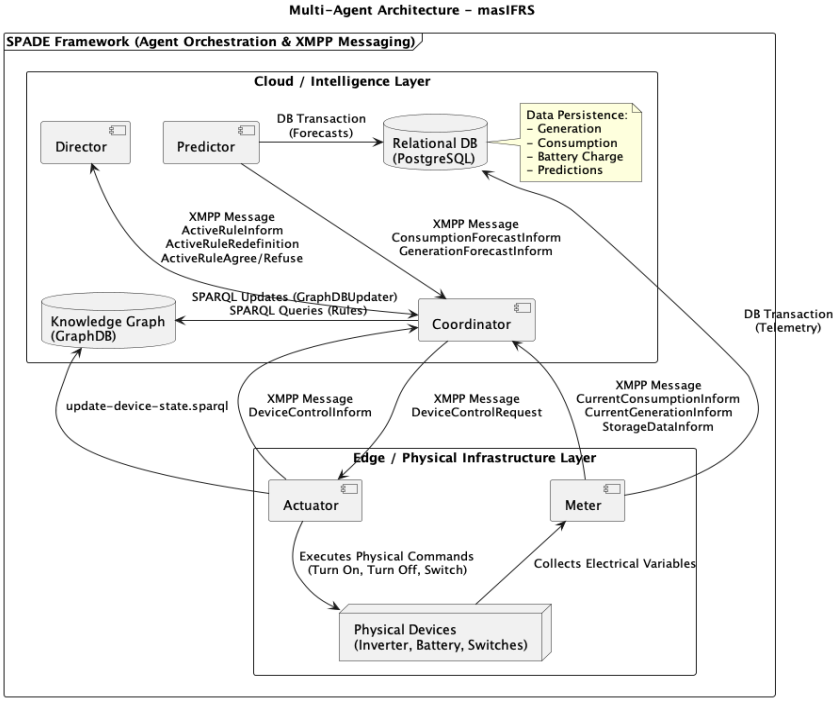

Figure 1: Architectural Model of masIFRS (source: the authors)

The Coordinator acts as the system's main reasoning unit. It does not interact directly with hardware but orchestrates the network. It assesses business rules using consumption, generation and prediction data provided by other agents. The Coordinator employs the GraphDBUpdater to merge measurements and update the triplestore via SPARQL. Based on the received data, it activates the operational rules to evaluate whether generation exceeds consumption or battery power is required. When a rule triggers, it formulates and sends precise instructions to the acting agents.

The Director operates at the supervision level, being tasked with dynamically managing and redefining the rules and policies that the Coordinator must follow. Its main attribution is to issue and

process communicative acts referring to active rules. The Director has the authority to restructure the logical behavior of the system at run time without reconfiguring the edge agents.

The Meter is an edge agent that solely collects data from physical sensors. It extracts the MG's electrical state and relays consumption, generation and prediction data to the Coordinator. The agent formats raw measurements into standardized messages, generating reports on current consumption, generation and battery states like load, capacity, and efficiency.

The Predictor is the agent in charge of data-driven intelligence and ML. It acts on demand, responding to requests with advance time series estimates. Using the ChronosForecaster model [12], it calculates one-hour forecast horizons for both the load demand and power generation from photovoltaic plants. Predictions are reported to the Coordinator for proactive decision-making.

The Actuator is responsible for translating the Coordinator's logical decisions into physical actions in the MG machinery. It is designed to receive structured commands and act on devices registered in the ontology, such as Inverters, Charge Controllers, and Smart Switches. The agent executes on/off or switching commands.

2.2. Semantic Models

The proposed framework uses ontologies and semantic web technologies to support interactions between agents, as well as their reasoning. One of the best practices regarding the development and use of ontologies is the reuse of previously defined knowledge in public and open access ontologies [13]. Therefore, we've decided to reuse IESO [2] for the representation of MG domain knowledge, as it aggregates diverse knowledge within the scope of electrical power systems, with a strong component developed for application in smart grids and MGs. To ensure interoperable and unambiguous communication between agents, the authors decided to develop an application ontology inspired by the Foundation for Intelligent Physical Agents' (FIPA) communicative acts.

2.2.1. IESO Modules

To instantiate the knowledge necessary for the operation of the proposed framework, the Actor Roles, Device, Measure, Commons, and Context-based Rules Matching Profile (CRMP) modules from the IESO [2] ontology were reused. The Actor Roles[3] module allows to instantiate actors, roles, and their behaviors. An *Actor* represents a stakeholder or entity and assumes at least one *Role*, while a *Role* performs at least one *Behavior*. It is suitable for the semantic description of the agents present in the framework, allowing their instantiation and the definition of behaviors according to predetermined interactions. The Device[4] module facilitates the semantic description of different types of devices, controllable or not, the functions they perform, the commands they accept, and their possible states. A *Device* is defined by one or more *Function*s and possible *State*s and may be composed of other devices or be part of one. A *Function* is characterized by having at least one *Command*, which, in turn, acts on the device's *State*. The Measure[5] module describes measurements, measurement values, type of measures, and units of measure, being useful to instantiate forecasted and device's data. A *Measurement* consists of a *Type of measure*, at least one *Measurement value*, and its *Unit of measure*. A *Type of measure* is also related to a *Unit of measure*, and a *Measurement value* is described by exactly one literal value and its *Unit of measurement*. This module eases the representation and conversion between measures of the same SI unit.

The IESO's Context-based Rules Matching Profile[6] (CRMP) module describes abstract rule-matching models based on contexts and associated profiles. A *Context* is composed of one or more *Profile*s and may be related to a specific time period. A *Profile*, in its most abstract form, is described by a value and associated unit, being extended to *BooleanProfile*, characterized by a boolean value; *LevelProfile*, which eases the definition of different levels of action, alert or notification; and *RangeProfile*,

[3]https://www.gecad.isep.ipp.pt/ieso/actor-roles/v1.1.0/
[4]https://www.gecad.isep.ipp.pt/ieso/device/v1.1.0/
[5]https://www.gecad.isep.ipp.pt/ieso/measure/v1.1.0/
[6]https://www.gecad.isep.ipp.pt/ieso/crmp/v1.1.0/

which defines a minimum and maximum limit for the value the profile can assume. A *Rule*, in turn, is described by a *Condition* and the *Action* to trigger if the condition is met. Currently, the *Condition* class is extended into three subclasses, namely: *UnaryOperatorCondition*, defined by an *Operand* and an *Operator*; *BinaryOperatorCondition*, described by two *Operand*s and an *Operator*; and *Composed-Condition*, which is composed by two or more *Condition*s. The *Action* class details specific actions that can be triggered by a *Rule*. It is extended to describe *ActuationAction*s on specific *Device*s; and *MonitoringNotificationAction*s for monitoring and notification sent to dashboards or send emails. CRMP must be extended to define the appropriate contexts and profiles for each rule. However, since rules can be independent of a specific *Context* or *Profile*, these classes' instantiation is not mandatory for reusing this module. After *Rule*s and respective *Action*s are instantiated, SPARQL queries ensure rule execution by interpreting the CRMP instances to validate each rule and returning a JavaScript Object Notation (JSON) string for the agent to act accordingly. In this way, the operating rules are no longer programmed into the agent, facilitating their updating without the need for recoding and recompiling the system. Subsection 2.3 details on how this module is used in the proposed framework.

2.2.2. Communicative Acts Ontology

To ensure a multi-agent interaction protocol within the proposed framework, we developed the Communicative Acts Ontology (CAO) application ontology. As an application-level ontology, CAO was developed solely for the purpose of ensuring the logic necessary for message exchange between agents, enabling semantic interoperability and reasoning about message content. Thus, the simplicity and straightforward perspective of the 101 (one-on-one) methodology [13] led us to use it in developing CAO. The 101 is an iterative process that continuously refines the ontology's requisites. CAO is publicly available on Zenodo (see file kb/communicative-acts.ttl) [15].

CAO is inspired by the performatives of FIPA's Communicative Act Library (ACL) Specification [16]. FIPA-ACL promotes interoperability by providing a set of communicative acts, facilitates their reuse, and provides a well-defined process for maintaining a set of communicative acts and act labels for use within the agent's communication language. Therefore, CAO only defines and extends the performatives present in the interaction protocols of the proposed masIFRS, namely: *Request*, *Inform*, *Agree* and *Refuse*. Each of these classes generically describes different types of messages exchanged between agents. CAO can be easily extensible to include other performatives.

The *Request* is extended to represent different request messages between different agents, such as the *CurrentConsumptionRequest*, *CurrentGenerationRequest*, *StorageDataRequest*, *ConsumptionForecastRequest*, *GenerationForecastRequest*, and *DeviceControlRequest*. The *Inform* class is extended to describe information messages, namely to respond to previous requests (i.e., *CurrentConsumptionInform*, *CurrentGenerationInform*, *StorageDataInform*, *ConsumptionForecastInform*, *GenerationForecastInform*, and *DeviceControlInform*) and to inform about the rule defined for the next period of operation (*ActiveRuleInform*) or redefine the rule to be executed (*ActiveRuleRedefinitionInform*). The *Agree* is extended to characterize agreed messages such as the *ActiveRuleAgree* and *ActiveRuleRedefinitionAgree* subclasses. Similarly, the *Refuse* class is extended to define the *ActiveRuleRefuse* subclass. Each subclass expresses the required knowledge for the agents to interpret each message and reason upon its content and fulfill their goals.

CAO ontology imports CRMP for exchanging messages about rules, which in turn imports the Measure module for representing measurable data, such as current consumption or consumption forecast. CAO allows agents to distinguish between an expected or unknown message before processing it. A simple ASK query allows the agent to understand whether the message type is known and should be processed, or not. If the message is of an expected type, a SELECT query extracts the necessary data as a JSON string, which the agent can transform into an object and use seamlessly. Otherwise, the agent simply replies that it is unaware of the type of message received. Agents use CONSTRUCT queries to generate messages to be sent to their peers, and SPARQL Update to insert and/or update data in their knowledge base, as will be demonstrated in section 3.

2.3. Reasoning

The Coordinator agent of masIFRS leverages semantic reasoning for the execution of business and operational rules. For this purpose, the CRMP module is used for the semantic description of business and operational rules, in combination with SPARQL for rule execution and decision-making. A key architectural advantage of this approach is the strict decoupling of operational logic from the agents' internal implementations. While it is technically possible to hardcode operational rules directly into the agents' behaviors, doing so tightly couples the execution logic with dynamic grid policies, requiring source code modifications and system downtime for every rule update. By defining rules as semantic instances and executing them via SPARQL queries interpreting the CRMP module, the system software remains completely agnostic to specific energy policies. This semantic decoupling allows the Director agent to dynamically redefine reasoning rules at runtime without reconfiguring or interrupting the physical edge agents. Furthermore, maintaining the logic within the knowledge graph enhances scalability and establishes a secure foundation for future integrations, where models can synthesize SPARQL policies without exposing or altering the core code of the autonomous entities.

The Coordinator agent is responsible for executing the rules that determine whether to charge the battery, discharge the battery, or draw power from the main grid in the next execution period. To achieve this goal, the following rules were defined (see file kb/business-rules-kb.ttl [15]):

$$if FG > FC : Battery_{charge}, \quad (1)$$

where FG and FC are the forecasted generation and consumption, respectively, for the next period. Equation 1 states that whenever the FG is higher than the FC, the surplus energy should be redirected to charge the battery. This is the *Generation Excess Rule*.

$$if FC > FG \wedge NP = night : Battery_{discharge}, \quad (2)$$

where NP is the next operating period. The *Night Deficit Rule*, determines that if the consumption forecast (FC) is greater than the generation forecast (FG) and the next operating period is during the night, then the charge available in the battery should be used to lower costs.

These rules are expressed semantically by the CRMP module and are part of the Coordinator agent's knowledge base. The *Generation Excess Rule* is a *Rule* having as premise a generation excess *BinaryOperatorCondition* and triggering a battery charge *ActuationAction*. A binary operator condition is composed of a binary operator ($>$) and a left (FG) and a right (FC) hand side operands. The *ActuationAction* identifies the *Device* to control ($Battery$) and the action to take ($charge$). The *Night Deficit Rule*, in turn, is a *Rule* having as premise a night deficit *ComposedCondition* while triggering a battery discharge *Action*. A composed condition is a condition composed by two or more conditions. In this case, it is composed of two *BinaryOperatorCondition*s, namely, a consumption excess condition ($FC > FG$) and a night shift condition ($NP = night$). Similarly, the *ActuationAction* identifies the *Device* to control ($Battery$) and the control action to take ($discharge$).

Having the rules expressed as ontology individuals, it is possible to write SPARQL queries for the types of conditions defined: *BinaryOperatorCondition* and *ComposedCondition*. Listings 1 and 2 present the SPARQL queries defined for each type of condition. It should be noted that these queries can be applied to any rule defined with the same type of conditions.

Listing 1. *BinaryOperatorCondition*-based rule SPARQL query.

```
PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX  ieso: <https://www.gecad.isep.ipp.pt/ieso/v1.1.0/>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT  ?dict
WHERE
  { ?rule     ieso:id                ?ruleId ;
              ieso:rulePremiseCondition  ?condition ;
              ieso:ruleTriggersAction  ?action .
    ?condition  ieso:id              ?conditionId ;
              ieso:description       ?conditionDescription ;
              ieso:leftHandSideOperand  ?leftOperand ;
              ieso:hasOperator       ?operator ;
              ieso:rightHandSideOperand  ?rightOperand .
```

```
15      ?leftOperand  ieso:id           ?leftOperandId ;
16                ieso:operandUnit      ?leftOperandUnit ;
17                ieso:operandValue     _:b0 .
18      _:b0      ieso:hasLiteralValue  ?leftOperandValue .
19      ?operator  ieso:id              ?operatorId ;
20                ieso:operatorSymbol   ?operatorSymbol .
21      ?rightOperand
22                ieso:id               ?rightOperandId ;
23                ieso:operandUnit      ?rightOperandUnit ;
24                ieso:operandValue     _:b1 .
25      _:b1      ieso:hasLiteralValue  ?rightOperandValue .
26      ?action   ieso:id               ?actionId ;
27                ieso:description      ?actionDescription ;
28                ieso:actionTriggeredByRule  ?rule ;
29                ieso:actuateOnDevice  ?device .
30      ?device   ieso:id               ?deviceId ;
31                ieso:isControlable    ?isControlable ;
32                ieso:hasState         _:b2 .
33      _:b2      ieso:stateStatus      ?stateStatus
34      BIND(if(( ( ( ?operatorId = "greater_than_operator"^^rdfs:Literal ) && ( ?op-
    eratorSymbol = ">"^^rdfs:Literal ) ) && ( strdt(str(?leftOperandValue), xsd:float)
    >  strdt(str(?rightOperandValue),  xsd:float)  )  ),  concat("{\"action_id\":\"",
    str(?actionId),      "\",\"action_description\":\"",      str(?actionDescription),
    "\",\"action_triggered_by_rule\":{\"rule_id\":\"", str(?ruleId), "\",\"rule_prem-
    ise_condition_description\":\"",     str(?conditionDescription),     "\"},\"actu-
    ate_on_device\":{\"device_id\":\"",   str(?deviceId),   "\",\"is_controlable\":",
    str(?isControlable),          ",\"current_state\":\"",          str(?stateStatus),
    "\",\"set_state\":\"", lcase(str(?actionDescription)), "\"}}"), "{}") AS ?dict)
      }
35
```

By observing Listing 1, it can be seen that the query selects only the variable `?dict` as a result (line 5). This variable is assigned by the BIND instruction (line 34), which generates a JSON string as a result of the rule's execution. From lines 7-9, the query selects individuals with a premise condition and a trigger action. Lines 10-14 ensure that the premise condition is defined by a left-hand side operand, a right-hand operand, and an operator. From lines 15-17, the query gets the left-hand side operand value and unit, and from lines 21-24 it selects the right-hand side operand value and unit. Lines 18-20 select the operator's symbol and id. This is crucial to ensure the proper rule execution. From lines 26-29, the query identifies the action to take if the rule's condition is true, and, from lines 30-33, it gets the device to be controlled. Finally, line 34 combines the BIND statement with an IF statement to generate the rule's outcome: a JSON string, using the CONCAT statement.

Listing 2. *ComposedCondition*-based rule SPARQL query.

```
01  PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
02  PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
03  PREFIX  ieso: <https://www.gecad.isep.ipp.pt/ieso/v1.1.0/>
04
05  SELECT  ?dict
06  WHERE
07    { ?rule     ieso:id               ?ruleId ;
08                ieso:rulePremiseCondition  ?condition ;
09                ieso:ruleTriggersAction  ?action .
10      ?condition  rdf:type            ieso:ComposedCondition ;
11                ieso:id               ?conditionId ;
12                ieso:description      ?conditionDescription
13      { SELECT  (GROUP_CONCAT(DISTINCT ?isTrue ; separator=', ') AS ?trigger)
14        WHERE
15          { ?condition  ieso:composedByCondition  ?c .
16            ?c        ieso:leftHandSideOperand  _:b0 .
17            _:b0      ieso:operandValue     _:b1 .
18            _:b1      ieso:hasLiteralValue  ?leftOperandValue .
19            ?c        ieso:hasOperator      _:b2 .
20            _:b2      ieso:id               ?operatorId .
21            ?c        ieso:rightHandSideOperand  _:b3 .
22            _:b3      ieso:operandValue     _:b4 .
23            _:b4      ieso:hasLiteralValue  ?rightOperandValue
24            BIND(if((   str(?operatorId)   =   "greater_than_operator"   ),   if((
    strdt(str(?leftOperandValue),   xsd:float)   >   strdt(str(?rightOperandValue),
    xsd:float) ), true, false), if(( str(?operatorId) = "equal_to_operator" ), if((
    str(?leftOperandValue) = str(?rightOperandValue) ), true, false), "unknown")) AS
```

```
    ?isTrue)
25          }
26        GROUP BY ?condition
27      }
28      ?action  ieso:id                ?actionId ;
29               ieso:description       ?actionDescription ;
30               ieso:actionTriggeredByRule  ?rule ;
31               ieso:actuateOnDevice   ?device .
32      ?device  ieso:id                ?deviceId ;
33               ieso:isControlable     ?isControlable ;
34               ieso:hasState          _:b5 .
35      _:b5     ieso:stateStatus       ?stateStatus
36      BIND(if((  str(?trigger)  =  "true"  ),  concat("{\"action_id\":\"",  str(?ac-
    tionId),   "\",\"action_description\":\"",   str(?actionDescription),   "\",\"ac-
    tion_triggered_by_rule\":{\"rule_id\":\"", str(?ruleId), "\",\"rule_premise_con-
    dition_description\":\"",    str(?conditionDescription),    "\"},\"actuate_on_de-
    vice\":{\"device_id\":\"", str(?deviceId), "\",\"is_controlable\":", str(?isCon-
    trolable),  ",\"current_state\":\"",  str(?stateStatus),  "\",\"set_state\":\"",
    lcase(str(?actionDescription)), "\"}}"), "{}") AS ?dict)
37    }
```

The query in Listing 2 also selects the `?dict` variable (line 5) that is instantiated by the BIND instruction on line 36. From lines 7-9, the query selects individuals with a premise condition and a trigger action, as in the previous rule. However, this query deals with rules whose premise is a composed condition, as can be seen from lines 10-12. Composed conditions, as the name indicates, are composed of two or more conditions. In our specific case, it is composed of two binary operand conditions, which facilitates the next step. From lines 13-26, a sub-query is defined with the GROUP_CONCAT and GROUP BY statements to select, verify, and return a boolean variable (`?trigger`) determining if the rule triggers the action, or not, according to the result of each binary operand condition. Looking at line 24, one can see the different IF statement which now considers not only the greater than operator but also the equals operator, given the defined condition (Eq. 2). Observing line 13, the DISTINCT statement ensures that if both conditions are true, only a "true" string is returned, meaning the rule triggered the action. From lines 28-35 the action and device to control are identified, and line 36 binds the result to the `?dict` variable, as in the previous rule.

In both rules, if the JSON string is an empty object ("`{}`"), it means that the rule does not trigger any action. The structure of the JSON object generated when the rules trigger actions will be demonstrated in the following section.

3. Demonstration

To demonstrate the application of ontologies for semantic interoperability and reasoning in agents, we prepared a simulation scenario using previously stored historical data from the Farroupilha campus of the Federal Institute of Rio Grande do Sul. For simplicity and ease of understanding for the reader, we decided to model only one of the available university campuses. Therefore, our simulation includes only one agent of each type, namely, a Director, a Coordinator, a Meter, a Predictor, and an Actuator. Due to limited space, we chose to focus the demonstration on the iterative process of prediction, rule execution, and action (if applicable), as it is sufficient to showcase the full range of semantic operations for agent interoperability and reasoning. It is important to note that, in the real-time execution of the system, this process depends on the data collected by the reading and database storage process.

The simulation scenario includes photovoltaic panels with a maximum generation capacity of 60 kWh, a battery with a maximum storage capacity of 60 kWh, and 8 classroom buildings. The date selected for the simulation is March 25, 2025. This choice is related to the fact that it is one of the days with the best quality of historical data collected. To ease the reader's follow along, the agents' input and output files are available online at [15]. Figure 2 shows the demonstration's workflow.

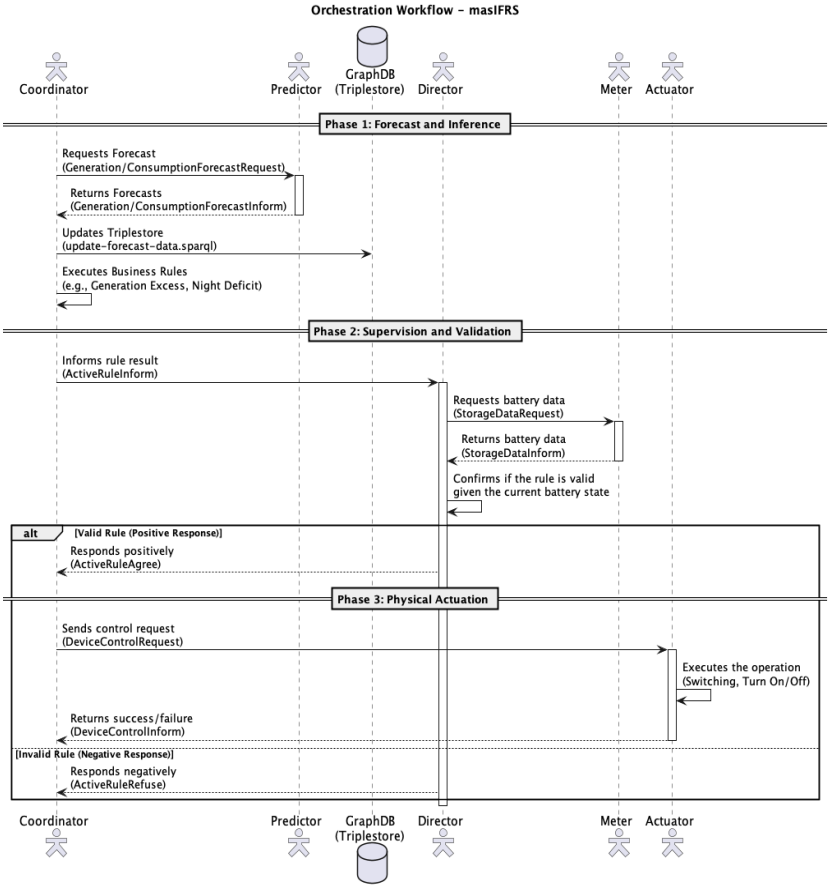

Figure 2: Demonstration workflow. (source: The authors)

The prediction, rule execution, and action process runs iteratively, and it (re)starts every 5 min before the execution period. The agent Coordinator starts each iteration by generating and sending consumption and generation forecast requests to the predictor agent. For this purpose, the Coordinator uses a CONSTRUCT query template (see files consumption-forecast-request.sparql and generation-forecast-request.sparql at sparql/coordinator/construct/ directory) to generate each type of request (i.e., consumption and generation). It is a template because it includes tags to be replaced at runtime to generate the query to be executed, namely the tags <[START]> and <[END]>. These tags allow the agent to determine the period for which it wants the prediction. The SPARQL files and respective request messages generated by the Coordinator are available in the directories results/coordinator/consumption_forecast_requests/ and results/coordinator/generation_forecast_requests/ at [15]. Listing 3 shows the consumption forecast request sent for the period 10:00-11:00.

Listing 3. Consumption forecast request message example.

```
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:ConsumptionForecastRequest ;
  time:hasBeggining [ a time:Instant ;
  time:inXSDDateTimeStamp "2025-03-25T10:00:00.000+00:00"^^xsd:DateTimeStamp ] ;
  time:hasEnd [ a time:Instant ;
  time:inXSDDateTimeStamp "2025-03-25T10:59:59.999+00:00"^^xsd:DateTimeStamp ] ;
  time:hasTime [ a time:Instant ;
  time:inXSDDateTimeStamp "2026-06-14T19:32:16.781845+00:00"^^xsd:dateTime ] .
```

After receiving the forecast requests, the predictor agent queries the message content (see files at sparql/predictor/select/ folder) to get the forecasting period (see directories get_consumption_forecast_request_data and get_generation_forecast_request_data in the folder results/predictor/), executes the forecasts using ChronosForecaster, and populates the respective SPARQL templates (see files at sparql/predictor/construct/ folder) to generate the responses to be sent to the Coordinator. Listing 4 displays an example of a consumption forecast inform message for the same period.

Listing 4. Consumption forecast inform message example.

```
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix ieso: <https://www.gecad.isep.ipp.pt/ieso/v1.1.0/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:ConsumptionForecastInform ;
    cao:consumptionForecast [ a ieso:Prediction ;
            ieso:hasMeasurement [ a ieso:Measurement ;
                    ieso:hasDateTime "2025-03-25T10:00:00.000+00:00"^^xsd:dateTime ;
                    ieso:hasMeasurementValue [ a ieso:MeasurementValue ;
                            ieso:hasLiteralValue "31.66"^^rdfs:Literal ;
                            ieso:hasUnitOfMeasure ieso:KilowattHour ] ] ] ;
 time:hasBeginning [ a time:Instant ;
 time:inXSDDateTimeStamp "2025-03-25T10:00:00.000+00:00"^^xsd:DateTimeStamp ] ;
 time:hasEnd [ a time:Instant ;
 time:inXSDDateTimeStamp "2025-03-25T10:59:59.999+00:00"^^xsd:DateTimeStamp ] ;
 time:hasTime [ a time:Instant ;
 time:inXSDDateTimeStamp "2026-06-14T19:32:16.930735+00:00"^^xsd:dateTime ] .
```

Similarly, the Coordinator queries the message content (see files get-consumption-forecast-inform-data.sparql and get-generation-forecast-inform-data.sparql at sparql/coordinator/select/directory) to obtain the forecast values (see directories get_consumption_forecast_inform_data and get_generation_forecast_inform_data in the folder results/coordinator/) and updates the knowledge base using a SPARQL Update template file (see file sparql/coordinator/update/update-forecast-data.sparql). The template tags (`<[CONSUMPTION_VALUE]>` and `<[GENERATION_ VALUE]>`) are replaced with the consumption and generation values, respectively. Before executing the rules, this agent runs a SPARQL Update file (sparql/coordinator/update/update-next-iteration-shift.sparql) to update the next iteration shift (i.e., "day" or "night"), due to the *Night Deficit Rule* (Eq. 2).

Following, the Coordinator runs both rules (see directory sparql/coordinator/rules/, or Listings 1 and 2). The implemented rules ensure that, at each iteration, only one of the rules can trigger an action. If any of these rules triggers an action (see the generation_excess_rule and night_deficit_rule folders in the /results/coordinator/ directory), the Coordinator uses a SPARQL CONSTRUCT template (see active-rule-inform.sparql in the folder /sparql/coordinator/construct/) to create an active rule inform message to be sent to the Director, where the tag `<[RULE_ID]>` is replaced by the rule's identifier (`ieso:id`). Listing 5 displays an example of the message sent to the Director.

Listing 5. Active rule inform message example.

```
@prefix br: <http://my.campus.org/business-rules#> .
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:ActiveRuleInform ;
  cao:activeRule br:generation_excess_rule ; time:hasTime [ a time:Instant ;
    time:inXSDDateTimeStamp "2026-06-14T19:32:17.476481+00:00"^^xsd:dateTime ] .
```

After receiving and querying (see /sparql/director/select/get-active-rule-inform-data.sparql) the Coordinator's message, the Director creates a storage data request (see storage-data-request.sparql in /sparql/coordinator/construct/ folder) to send to the Meter agent. The Director uses the storage data to determine if he accepts or rejects the rule proposed by the Coordinator for the next period. Listing 6 presents an example of the storage request message (files available in the directory /results/director/storage_data_request/ only for the periods that some rule triggers an action).

Listing 6. Storage data request message example.

```
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:StorageDataRequest ;
  time:hasTime [ a time:Instant ; time:inXSDDateTimeStamp "2026-06-
14T19:32:17.521939+00:00"^^xsd:dateTime ] .
```

As before, the Meter agent queries the message (see file get-storage-data-request-data.sparql available at /sparql/meter/select/ folder) to identify the request and uses a SPARQL template to con-

struct the response message (see file storage-data-inform.sparql at /sparql/meter/construct/ directory) with the storage data for the Director. Listing 7 shows an example of a storage data inform message (full files available at /results/meter/storage_data_inform/ folder).

Listing 7. Storage data inform message example.

```
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix ieso: <https://www.gecad.isep.ipp.pt/ieso/v1.1.0/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:StorageDataInform ;
    cao:storageCapacity [ a ieso:Energy ;
            ieso:hasDateTime "2025-03-25T10:00:00.000+00:00"^^xsd:dateTime ;
            ieso:hasMeasurementValue [ a ieso:MeasurementValue ;
                    ieso:hasLiteralValue "60"^^rdfs:Literal ;
                    ieso:hasUnitOfMeasure ieso:KilowattHour ] ] ;
    cao:storageCurrentCharge [ a ieso:Energy ;
            ieso:hasDateTime "2025-03-25T10:00:00.000+00:00"^^xsd:dateTime ;
            ieso:hasMeasurementValue [ a ieso:MeasurementValue ;
                    ieso:hasLiteralValue "13.5"^^rdfs:Literal ;
                    ieso:hasUnitOfMeasure ieso:KilowattHour ] ] ;
    cao:storageEfficiency [ a ieso:Measurement ;
            ieso:hasDateTime "2025-03-25T10:00:00.000+00:00"^^xsd:dateTime ;
            ieso:hasMeasurementValue [ a ieso:MeasurementValue ;
                    ieso:hasLiteralValue "0.9"^^rdfs:Literal ;
                    ieso:hasUnitOfMeasure ieso:Percent ] ] ;
    time:hasBeginning [ a time:Instant ;
            time:inXSDDateTimeStamp                                    "2025-03-
25T10:00:00.000+00:00"^^xsd:DateTimeStamp ] ;
time:hasEnd     [     a     time:Instant     ;     time:inXSDDateTimeStamp     "2025-03-
25T10:59:59.999+00:00"^^xsd:DateTimeStamp ] ;
time:hasTime     [     a     time:Instant     ;     time:inXSDDateTimeStamp     "2026-06-
14T19:32:17.584117+00:00"^^xsd:dateTime ] .
```

With this data, the Director computes the maximum possible charge/discharge state at the end of the next period to determine if he accepts or refuses the rule. In case of discharge, the Director ensures the battery charge is below 95%, while in case of discharge it must be above 20%. According to this, the agent creates an answer to send to the Coordinator (see /sparql/director/construct/ directory). Listings 8 e 9 illustrate examples of accept and refuse messages, respectively (see folders active_rule_agree/ and active_rule_refuse/ within /results/director/).

Listing 8. Active rule agree message example.

```
@prefix br: <http://my.campus.org/business-rules#> .
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:ActiveRuleAgree ;
    cao:agreeWithCommunicativeAct br:generation_excess_rule ;
    time:hasTime [ a time:Instant ;
    time:inXSDDateTimeStamp "2026-06-14T19:32:17.774400+00:00"^^xsd:dateTime ] .
```

Listing 9. Active rule refuse message example.

```
@prefix br: <http://my.campus.org/business-rules#> .
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:ActiveRuleRefuse ;
cao:refuseCommunicativeAct br:night_deficit_rule ;time:hasTime [a time:Instant ;
    time:inXSDDateTimeStamp "2026-06-14T19:32:31.729541+00:00"^^xsd:dateTime ] .
```

Upon receiving the response, the Coordinator queries it (see folder /sparql/coordinator/select/, files get-active-rule-agree-data.sparql and get-active-rule-refuse-data.sparql). If the Director agrees with the proposed rule, the Coordinator creates a device control request message to send to the Actuator agent using a SPARQL template file, where the tags are replaced by the action and device identifiers (see file device-control-request.sparql in folder /sparql/coordinator/construct/). Listing 10 presents an example of a device control request message (see /results/coordinator/device_control_request/).

Listing 10. Device control request message example.

```
@prefix br: <http://my.campus.org/business-rules#> .
@prefix cao: <http://my.campus.org/communicative-acts#> .
@prefix time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a cao:DeviceControlRequest ;cao:controlAction br:battery_charge_action ;
    cao:controlDevice br:stationary_battery ;time:hasTime [ a time:Instant ;
    time:inXSDDateTimeStamp "2026-06-14T19:32:17.817114+00:00"^^xsd:dateTime ] .
```

The Actuator agent queries the received message (see /sparql/actuator/select/device-control-request-data.json) to identify the target device and the specific action to be performed (see /results/actuator/get_device_control_request_data/ directory). Finally, after operation, it constructs a response message (see file /sparql/actuator/construct/device-control-inform.sparql) to send to the Coordinator informing about the success or failure of the control operation. Examples of device control inform messages are available in the /results/actuator/device-control-inform/ folder at [15].

Table 1 presents the hourly results of the agents' operations during the simulation.

Table 1

Hourly results of the agents' operations.

Perido	CF (kWh)	GF (kWh)	Rule	Battery charge	Charge after rule execution	Decision	Control
0	13.4	0	*Night Deficit*	57.3	50	*Agree*	*Success*
1	16.47	0	*Night Deficit*	50	42.7	*Agree*	*Success*
2	15.89	0	*Night Deficit*	42.7	35.4	*Agree*	*Success*
3	16.5	0	*Night Deficit*	35.4	28.1	*Agree*	*Success*
4	16.67	0	*Night Deficit*	28.1	20.8	*Agree*	*Success*
5	16.2	0	*Night Deficit*	20.8	13.5	*Agree*	*Success*
6	18.06	0.75	*Night Deficit*	13.5	6.2	*Refuse*	n.a.
7	19.9	5.37	*Night Deficit*	13.5	6.2	*Refuse*	n.a.
8	35.06	15.47	n.a.	n.a	n.a.	n.a.	n.a.
9	31.94	10.84	n.a.	n.a.	n.a.	n.a	n.a.
10	31.66	32.29	*Generation Excess*	13.5	20.8	*Agree*	*Success*
11	34.86	29.36	n.a.	20.8	n.a.	n.a.	n.a.
12	11.49	36.78	*Generation Excess*	20.8	28.1	*Agree*	*Success*
13	38.28	38.24	n.a.	28.1	n.a.	n.a.	n.a.
14	44.36	30.72	n.a.	28.1	n.a.	n.a.	n.a.
15	49.1	17.77	n.a.	28.1	n.a.	n.a.	n.a.
16	46.35	8.43	n.a.	28.1	n.a.	n.a.	n.a.
17	40.71	4.39	n.a.	28.1	n.a.	n.a.	n.a.
18	37.81	0.45	n.a.	28.1	n.a.	n.a.	n.a.
19	50.49	0	*Night Deficit*	28.1	20.8	*Agree*	*Success*

Perido	CF (kWh)	GF (kWh)	Rule	Battery charge	Charge after rule execution	Decision	Control
20	41.89	0	*Night Deficit*	20.8	13.5	*Agree*	*Success*
21	33.88	0	*Night Deficit*	13.5	6.2	*Refuse*	n.a.
22	20.28	0	*Night Deficit*	13.5	6.2	*Refuse*	n.a.
23	17.58	0	*Night Deficit*	13.5	6.2	*Refuse*	n.a.

For each simulation period, it can be seen the consumption and generation forecasts, the triggered rule, the battery charge and charge after executing the rule, the Director's decision and the status of the control action. It should be noted that when no rule is launched, the Director does not request battery data or calculate the state of charge after executing the rule. Consequently, there is no request or control of the device. This is also the case when the Director refuses the rule. Analyzing the results obtained, we verified that the proposed system behaved as expected.

4. Conclusions

The evaluation of the masIFRS shows that the hypothesis investigated in this study is corroborated. We conclude that the proposed model achieves a solid separation between business logic and agent implementation. Thus, the software structure will remain agnostic to dispatch and energy control policies, encapsulating this operational rationale only in queries and semantic rules written in SPARQL. Furthermore, this provides strong scalability for the system, enabling the overall management of MGs to be updated and reconfigured at runtime within the architecture of its knowledge graph, without structural changes or recompilation of the source code for autonomous entities.

Future work aimed at the continuous improvement of this Energy Management System includes the use of large language models (LLMs) in the semantic knowledge base. The key aspect of this expansion will be to dynamically generate SPARQL rules for MAS using LLMs and to utilize the LlamaIndex [17] framework as a data-orchestration and context-indexing mechanism. This methodological evolution will allow the Director agent, or the supervisory system itself, to automatically infer, at runtime, new energy management policies from abstract operational needs expressed in natural language and produce the logical constraints of an ontology.

Furthermore, future contributions will integrate Federated Learning approaches into the prediction and decision-making routines. This collaborative method enables decentralized training, ensuring the privacy and security of consumption and operational data generated by campus buildings [18, 19]. Federated Learning prevents sensitive consumer information from propagating across the network because each physical agent strictly localizes it. This will be the solution of choice for resolving the tension between the need to guarantee privacy and data preservation and the need for global analytical training and inference, which require scalable and accurate models.

Declaration on Generative AI

During the preparation of this work, the authors used NotebookLM, DeepL, and Grammarly to check references, translate text, and check grammar and spelling. The authors also used NotebookLM to correct the .puml scripts for Figures 1 and 2 used in the UML diagram editor. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] O. P. Mahela, M. Khosravy, N. Gupta, B. Khan, H. H. Alhelou, R. Mahla, N. Patel, and P. Siano, "Comprehensive overview of multi-agent systems for controlling smart grids," CSEE J. Power Energy Syst., vol. 8, no. 1, pp. 115–131, 2022, doi: 10.17775/CSEEJPES.2020.03390.

[2] Santos G, Morais H, Pinto T, Corcha JM, Vale Z. Intelligent energy systems ontology to support markets and power systems co-simulation interoperability. Energy Conversion and Management: X [Internet]. 2023 Nov; Available from: https://doi.org/10.1016/j.ecmx.2023.100495.
[3] A. R. Singh, R. Seshu Kumar, M. Bajaj, C. B. Khadse, and I. Zaitsev, "Machine learning-based energy management and power forecasting in grid-connected microgrids with multiple distributed energy sources," Sci. Rep., vol. 14, Art. no. 19207, 2024, doi: 10.1038/s41598-024-70336-3.
[4] W. P. Arévalo, D. Ochoa-Correa, E. Villa-Ávila, V. Iñiguez-Morán, and P. Astudillo-Salinas, "Systematic review of hierarchical and multi-agent optimization strategies for P2P energy management and electric machines in microgrids," Appl. Sci., vol. 15, no. 9, Art. no. 4817, 2025, doi: 10.3390/app15094817.
[5] O. P. Mahela, M. Khosravy, N. Gupta, B. Khan, H. H. Alhelou, R. Mahla, N. Patel, and P. Siano, "Comprehensive Overview of Multi-agent Systems for Controlling Smart Grids," CSEE Journal of Power and Energy Systems, Dec. 2020, doi: 10.17775/CSEEJPES.2020.03390.
[6] Gillani, S., Laforest, F., & Picard, G. (2014). A Generic Ontology for Prosumer-Oriented Smart Grid. Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference.
[7] Hamidi, M., Raihani, A., & Bouattane, O. (2023). Sustainable Intelligent Energy Management System for Microgrid Using Multi-Agent Systems: A Case Study. Sustainability, 15, 12546.
[8] Lee, S., Seon, J., Hwang, B., Kim, S., Sun, Y., & Kim, J. (2024). Recent Trends and Issues of Energy Management Systems Using Machine Learning. Energies, 17, 624.
[9] Li, Y., He, S., Li, Y., Shi, Y., & Zeng, Z. (2023). Federated Multiagent Deep Reinforcement Learning Approach via Physics-Informed Reward for Multimicrogrid Energy Management. IEEE Transactions on Neural Networks and Learning Systems.
[10] Wang, Q., Wang, Y., Chen, Z., & Soares, J. (2024). Multi-agent system consistency-based cooperative scheduling strategy of regional integrated energy system. Energy, 295, 130904
[11] Yao, R., Hu, Y., & Varga, L. (2023). Applications of Agent-Based Methods in Multi-Energy Systems—A Systematic Literature Review. Energies, 16, 2456.
[12] Fatir Ansari, A., "Chronos: Learning the Language of Time Series", arXiv e-prints, Art. no. arXiv:2403.07815, 2024. doi:10.48550/arXiv.2403.07815.
[13] Noy, N. & Mcguinness, D. (2001). Ontology Development 101: A Guide to Creating Your First Ontology. Knowledge Systems Laboratory. 32. URL: https://www.researchgate.net/publication/243772462_Ontology_Development_101_A_Guide_to_Creating_Your_First_Ontology
[14] SPARQL 1.1 Query Language. URL: https://www.w3.org/TR/sparql11-query/
[15] G. Santos and R. A. Manfredini, "Demonstration of a Multi-Agent Ontology-based Architecture for Semantic Interoperability and Reasoning in Microgrid Energy Management". Zenodo, Jul. 26, 2026. doi: 10.5281/zenodo.21598072.
[16] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification (2001). URL: https://jmvidal.cse.sc.edu/library/XC00037H.pdf
[17] LlamaIndex Framework. URL: https://developers.llamaindex.ai/python/framework/
[18] Y. Li, S. He, Y. Li, Y. Shi and Z. Zeng, "Federated Multiagent Deep Reinforcement Learning Approach via Physics-Informed Reward for Multimicrogrid Energy Management," in IEEE Transactions on Neural Networks and Learning Systems, vol. 35, no. 5, pp. 5902-5914, May 2024, doi: 10.1109/TNNLS.2022.3232630.
[19] S. Lee and D.-H. Choi, "Federated reinforcement learning for energy management of multiple smart homes with distributed energy resources," IEEE Transactions on Industrial Informatics, vol. 18, no. 1, pp. 488–497, Jan. 2022.