On Sound Experiment Execution with Learning Agents in Cyber-Physical Energy Systems

Eric MSP Veith[1,2] Stephan Balduin[2] Arlena Wellßow[1] Torben Logemann[1]

[1]Carl von Ossietzky University Oldenburg
Research Group Adversarial Resilience Learning
Oldenburg, Germany
Email: firstname.lastname@uol.de

[2]OFFIS – Institute for Information Technology
Research Group Power Systems Intelligence
Oldenburg, Germany
Email: firstname.lastname@offis.de

***Abstract*—Autonomous and learning systems, such as Multi-Agent Systems or learning agents based on Deep Reinforcment Learning, has firmly established themselves as a foundation for approaches to create resilient and efficient Cyber-Physical Energy Systems. A substantial amount of research into different aspects of these systems is backed by simulation. However, the presentation of the simulation setup, experiment design, and experiment results evaluation often lacks crucial information, making it hard to reproduce or compare to other researcher's results. In this paper, we present the experiment design tooling of *arsenAI*, a part of the *palaestrAI* software ecosystem. We describe the work in progress on the experiment definition and mechanisms in place to aid in sound and reproducible experimentation with learning agents in co-simulated Cyber-Physical Energy Systems. We provide a document schema, name necessary tools, and also describe the relevant building blocks that enable reproducible experimentation with learning agent systems in co-simulated Cyber-Physical Energy Systems.**

***Keywords*—agent systems; learning agents; reinforcement learning; complex co-simulation; cyber-physical systems; modelling and simulation**

I. Introduction and Related Work

Agent systems are well-known for many aspects of research into Cyber-Physical Energy Systems (CPESs), and learning agents—e. g., such based on Deep Reinforcment Learning (DRL)—have established a firm foothold in the domain as well. From the hallmark paper that introduced DRL [1] to the development of MuZero [2] and AlphaStar [3], learning agents research has inspired many applications in the energy domain, including real power management [4], reactive power management and voltage control [5], [6], black start [7], anomaly detection [8], or analysis of potential attack vectors [9], [10].

It is in the very nature of especially learning agent systems that the most common way to demonstrate advances in the domain is through experimentation. Published papers present key performance indicators from the domain, such as a plot of voltage over time. To show that an agent learns, a plot of the reward function or utility function over time (or aggregated over training episodes) is depicted, alongside tables with cumulative values.

However, in many cases, crucial features are missing that would allow to reproduce these experiments. Basic values such as the initial seed for random number generators or the software packages, together with their version numbers, are usually not specified. In order to be meaningful, experiments would be constructed starting from a hypothesis, with invariants to validate or refute it. A Design of Experiments (DoE) [11] approach would allow to specify parameters and factors; the latter one to be varied in order to gauge the influence of particular input variables on the presented agent's or algorithm's general performance. In the context of CPESs, there is usually a number of such factors that can be considered, such as weather data, time of the year, node placement, or inverter capabilities.

In general, the goal for sound experimentation would be both, reproducibility, and a concise way to specify an experiment. CPES simulation often happens as co-simulation, using frameworks such as *mosaik* [12] or *CPSWT/C2WT-TE* [13]. Of those, some allow scenario-based modelling [14], but usually, the co-simulation itself does not consider the experiment stage, only the simulation execution stage, which follows a separation-of-concerns idea. As such, the concern for sound experimentation falls to other software packages. Surprisingly, even though there are libraries that provide the basic facilities for DoE, there is no tool suite available that would ensure sound experimentation with learning agents in such a co-simulation.

In this paper, we present the work-in-progress state of the *palaestrAI* software suite. This suite comprises a number of packages whose goal is to enable researchers to conduct reproducible and reliable experiments with (learning) agents in complex, co-simulated Cyber-Physical Systems (CPSs). The software suite consists of four major parts: *arsenAI*, *palaestrAI* itself, *hARL*, and *palaestrAI Environments*. In this stack, arsenAI is responsible to read experiment definitions, evaluate DoE statements, and create the concrete experiment run definitions. These experiment run definitions are instantiations of an experiment design, i. e., all factors are set to concrete values. *palaestrAI* itself takes care of proper execution. The *hARL* package offers implementation of learning agents, e. g., DRL algorithms such as Proximal Policy Gradient (PPO) or Soft Actor Critic (SAC), or the Adversarial Resilience Learning (ARL) agent reference implementation. *palaestrAI Environments* finally provides a unified interface to co-simulated environments, e. g., the MIDAS power grid reference scenario [15] through mosaik [12]. The complete software stack, organized in terms of a typical experimentation workflow, is depicted in Figure 1.

 ISBN: 978-1-68558-139-8

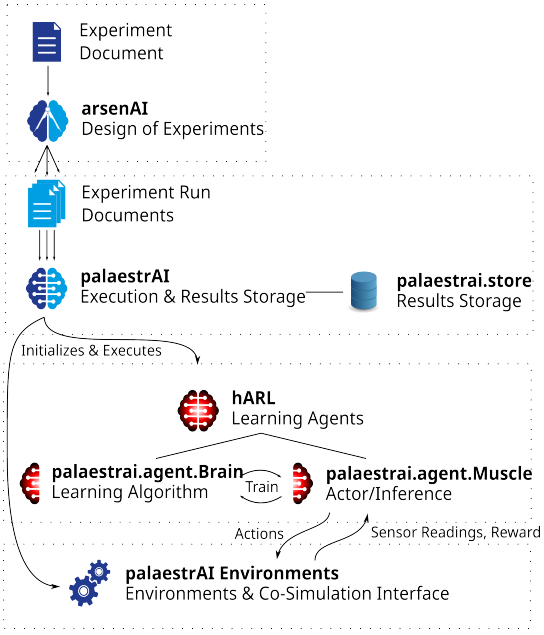

Figure 1. The palaestrAI software stack [16]

The remainder of this paper is structured as follows: In Section II, we present the experiment definition document proper and describe its schema and design goals. Section III describes all features that aid in the reproducibility of experiments conducted with palaestrAI/arsenAI. Section IV presents methods built into the software stack to evaluate results from conducted experiments. We conclude in Section V with an outlook on our current work.

II. Experiment Definition Document

An experiment definition document serves as an intermediary data format, which needs to be both human-readable/-writable as well as machine-readable. Well-established formats at this interface are eXtensible Markup Language (XML), JavaScript Object Notation (JSON), and YAML Ain't A Markup Language (YAML). Since the barrier for humans should be as low as possible, YAML was chosen for arsenAI. YAML can be structurally validated using *YAML Schema*, which ensures the syntactic correctness of the document.

Experiment design for co-simulation setups is the major premiss for this document. This means that it specifies software modules, such as particular agent implementations or environments, which are used for the co-simulation. Co-simulation setups are not monolithic and no assumptions can be made with regards to the simulators that will be used. Thus, the experiment document itself cannot specify behaviors; it can only specify software modules and parameters to it. In this sense, parameters are what defines their behavior and not a runtime configuration, such as logging. This also means that the experimenter has to trust the software to behave correctly.

Scripted "events" can be part of an experiment design (sometimes called *incursions*). Those are then software modules, too (albeit provided by the experimenter), which use the general co-simulation Application Programming Interface (API) and are written with the particular data exchange of concrete simulators in mind.

An important part of experiment design is the ability to reason about interactions between different parts of the setup and gauging the influence of certain factors in order to validate a hypothesis. For example, an experiment would try to answer the question "how does the choice of a particular agent training algorithm influence the observed performance of a variable over time," and explicitly not "run a simulation with agent A in environment E with configuration C and observe state variable X." As such, an experiment introduces a meta-layer. The DoE paradigm introduces *factors*, whose influence should be observed, and *parameters*, which represent values that are fixed for an experiment. The variation of factors and gauging of their influence is the core idea of DoE.

A concrete instantiation of all factors in an experiment definition creates an *experiment run definition*, which can then be picked up by a software and executed. Each experiment consists of at least one *phase*: A phase designates a particular stage of execution within an experiment, with concrete agents, each having a concrete set of sensors and actuators assigned, acting in co-simulated environments. Each phase has its own *termination condition*, each of which is an invariant against which the current state of agents and environments is checked during runtime to determine whether the phase has ended. Termination conditions can be based on the state of an environment (e.g., a blackout can be a termination condition for a power grid environment) or the state of an agent (e.g., its performance metric can hit a certain threshold). Phases can be repeated several times (called an *episode*), which is useful for learning agents. For example, an untrained agent may trigger an environment termination condition several times before developing a sensible strategy; thus, automatic restarts are necessary.

Subsequent phases can switch parts of the configuration, such as the agents that participate in it, their hyperparameters, termination conditions, and so an. A typical use case for several phases would be a training phase followed by a subsequent testing phase.

Here, the DoE approach allows to experimentally investigate hypothesis that are typical for learning agents in CPESs. For example, in order to evaluate whether an agent would learn a robust voltage control scheme, it would have to be tested in different power grid layouts, in different usage scenarios, even under adverse conditions. Also, learning agents need to be compared in terms of the algorithms they use and their respective hyperparameters. The search grid of combinations can be automatically generated through the DoE services that software packages such as arsenAI offer; even autocurriculum setups with different agents training each other or competition-style comparison setups can easily be realized with way.

An experiment document first independently provides a number of definitions: *environments*, *agents*, named sets of *sensors* and *actuators*, *simulation execution strategies* including termination conditions, and *phase* configurations (e.g., training or testing mode, or the number of episodes). An example for

 ISBN: 978-1-68558-139-8

these definitions can be found in Figure 2. Afterwards, concrete phases are listed in terms of a phase schedule. Each phase lists the desired factors under investigation and, thus, designs the search grid. Figure 3 is an example of such a schedule (and a continuation of Figure 2).

Finally, each experiment document requires a user-defined name. In addition, it allows to specify a random seed for reproducibility.

A design goal for the experiment document format described here is the ease-of-use for humans. Specifically, verbosity should be reduced in order to increase conciseness and, therefore, the expressiveness of an experiment document. The schedule configuration is the target for this goal, as it usually contains a number of repetitions. For example, the agents participating a specific phase may change, but the configuration of the environment might stay the same. In this case, simple repeating the environment configuration will make it difficult to discern actual changes from definitions copied over from the previous phase. Here, a cascading definition—similar to Cascading Style Sheets (CSS)—is employed: Any definition (agents, environments, or phase configuration) that is not explicitly given is considered to be equal to the one from the previous phase. Continuing with the example given, an environment has then to be defined only once (in the first phase it is used), and omitting it afterwards implicitly means that it is re-used as-is in any following phase that does not redefine the *environments* definition part of the schedule. Figure 4 lists the algorithm that is used to implement this cascading configuration scheme.

The example shown in Figure 2 defines the major components of an autocurriculum-type training and test of agents. Here, four agents should be trained, two power grid operator agents and two adversary ("attacker") agents. For easier reference, they are dubbed "Gandalf" and "Sauron" respectively. Of the two pairs, one instance should be trained in autocurriculum fashion (i. e., yielding one "Gandalf vs. Sauron" training), and two without adversary. The hypothesis that should be verified with this setup is: Autocurriculum-trained agents perform better (at the task of voltage control) than those that train alone. The task (i. e., voltage control through reactive power) is learned; the agent's utility functions are given as their *objective*. Agents are comprised of a learning module (nicknamed *Brain* in palaestrAI parley) and rollout workers (called *Muscles*). Each con receive their own set of parameters. In Figure 2, only a configuration for SAC is shown due to space constraints.

Figure 3 then defines the training and subsequent testing phases.

III. Reproducibility

Reproducibility is an important aspect of simulation: It allows other researchers to reproduce the same experiment, arrive at the same results data and, therefore, verify the conclusions that were drawn from it. Especially Artificial Intelligence (AI) researchers have been under scrutiny because of reproducibility issues of many publications in the past [17]. Ideally, one would only need to distribute an experiment

```yaml
uid: Classic ARL
seed: 2022
version: 3.5.0
output: palaestrai-runfiles
repetitions: 1
max_runs: 300
definitions:
  environments:
    midasmv_tar_ms:
      environment:
        name: MosaikEnvironment
        uid: midas_powergrid
        params: {}
      reward:
        name: ExtendedGridHealthReward
  agents:
    gandalf_sac_single:
      name: Gandalf SAC (single-agent-training)
      brain: &sac_brain
        name: harl:SACBrain
        params:
          fc_dims: [48, 48]
          update_after: 1000
          batch_size: 500
          update_every: 200
      muscle: &sac_muscle
        name: harl:SACMuscle
        params: {}
      objective: &defender_objective
        name: ArlDefenderObjective
        params: {}
    gandalf_sac_ac:
      name: Gandalf SAC (autocurriculum-training)
      brain: *sac_brain
      muscle: *sac_muscle
      objective: *defender_objective
    sauron_sac_single:
      name: Sauron SAC (single-agent-training)
      brain: *sac_brain
      muscle: *sac_muscle
      objective: &attacker_objective
        name: ArlAttackerObjective
        params: {}
    sauron_sac_ac:
      name: Sauron SAC (autocurriculum-training)
      brain: *sac_brain
      muscle: *sac_muscle
      objective: *attacker_objective
  sensors:
    all_sensors:
      midas_powergrid: [s1, s2]
  actuators:
    attacker_actuators:
      midas_powergrid: [a1, a2]
    defender_actuators:
      midas_powergrid: [a3, a4]
  simulation:
    vanilla_sim:
      name: TakingTurns
      conditions:
        - name: EnvTerminates
          params: {}
  phase_config:
    train: {mode: train, worker: 1, episodes: 10}
    test: {mode: test, worker: 1, episodes: 3}
```

Figure 2. Example of the definitions part of an experiment document (identifiers shortened for readability)

 ISBN: 978-1-68558-139-8

```
schedule:
  - Adversary Single Training:
      environments: [[midasmv_tar_ms]]
      agents: [[sauron_sac_single]]
      simulation: [vanilla_sim]
      phase_config: [training]
      sensors: &sensors_single
        sauron_sac_single: [all_sensors]
        gandalf_ddpg_single: [all_sensors]
        gandalf_sac_single: [all_sensors]
      actuators: &actuators_single
        sauron_sac_single: [attacker_actuators]
        gandalf_sac_single: [defender_actuators]
  - Operator Single Training:
      environments: [[midasmv_tar_ms]]
      agents: [[gandalf_sac_single]]
      simulation: [vanilla_sim]
      phase_config: [training]
      sensors: *sensors_single
      actuators: *actuators_single
  - Autocurriculum Training:
      environments: [[midasmv_tar_ms]]
      agents: [[sauron_sac_ac, gandalf_sac_ac]]
      simulation: [vanilla_sim]
      phase_config: [training]
      sensors: &sensors_ac
        sauron_sac_ac: [all_sensors]
        gandalf_sac_ac: [all_sensors]
      actuators: &actuators_ac
        sauron_sac_ac: [attacker_actuators]
        gandalf_sac_ac: [defender_actuators]
  - Adversary (S) vs. Operator (AC) Test:
      environments: [[midasmv_tar_ms]]
      agents: [
        [sauron_sac_single, gandalf_sac_ac]]
      simulation: [vanilla_sim]
      phase_config: [test]
      sensors:
        <<: [*sensors_ac, *sensors_single]
      actuators:
        <<: [*actuators_ac, *actuators_single]
  - Adversary (AC) vs. Operator (S) Test:
      environments: [[midasmv_tar_ms]]
      agents: [
        [sauron_sac_ac, gandalf_sac_single]]
      simulation: [vanilla_sim]
      phase_config: [test]
      sensors:
        <<: [*sensors_ac, *sensors_single]
      actuators:
        <<: [*actuators_ac, *actuators_single]
  - Adversary (AC) vs. Operator (AC) Test:
      environments: [[midasmv_tar_ms]]
      agents: [[sauron_sac_ac, gandalf_sac_ac]]
      simulation: [vanilla_sim]
      phase_config: [test]
      sensors: *sensors_ac
      actuators: *actuators_ac
```

Figure 3. Experiment schedule with factors definition (continued from Figure 2

document and all non-public data in order to allow others to recreate an experiment; distributing full setups, up to fully-configured virtual machine images, should not be necessary. Since co-simulation setups with learning agents are even more complex, ensuring reproducibility requires a number of additional precautions.

The previous section already introduced the notion of a specific seed; seeding Pseudo Random-Number Generator (PRNG) with a known number is a common practice. Then,

function EXPAND-SCHEDULE(experimentrun)
  $schedule = \text{EMPTY-LIST}$
  for $phase \in experimentrun.phases$ **do**
    $schedule \leftarrow schedule \cup \text{DEEP-COPY}($
      $\text{UPDATE-MAPPING}(schedule, phase))$
  end for
  return $schedule$
end function
function UPDATE-MAPPING(src, upd)
  for $key, value \in upd$ **do**
    if val isa $Mapping$ **then**
      $entry \leftarrow val_{key} \vee \text{EMPTY-MAPPING}()$
      $src_{key} \leftarrow \text{UPDATE-MAPPING}(entry, value)$
    else
      $src_{key} \leftarrow value$
    end if
  end for
  return src
end function

Figure 4. Phase configuration cascade algorithm

the PRNGs will still emit random numbers, but their sequence will stay the same. This is an obviously important feature for reproducibility. Software packages usually allow this seeding with custom values, e. g., *NumPy*, a commonly used library in scientific software, has a chapter in their documentation about seeding; *PyTorch*, a popular deep learning library, allows this equally.

Since the premiss of co-simulation DoE is that software packages are used transparently, at least ensuring that the same software versions are used across devices is another important part of reproducibility. Almost all programming languages allow to query the software packages that are currently installed. For example, in Python, the command `pip freeze` outputs a list that can later be used to re-install the same versions. Such a `requirements.txt`-style software package versions list can be embedded in the experiment (run) document. Especially noteworthy in this regard are source code management systems like *git* that allow to unambiguously identify each state of a software repository through hashed commits. An additional section `software` (not shown in the abbreviated Figures 2 and 3 due to space constraints) contains a mapping of package management identifier to software package specfications list. For example, one package management identifier would be `pip3` to denote Python pacakges; it maps to a list of `requirements.txt`-style software package specifications that would be passed verbatim to `pip3 install`.

Finally, the experiment and experiment run documents themselves need to be checked for their identity: Changes to such a document should be detected by the software without requiring the user to change its name or any other unique means of identification: A human can easily forget to change an identifier when a factor is changed. Moreover, when convenience features are being used (e. g., the cascading property of arsenAI's experiment (run) documents, or YAML anchors), then the document becomes semantically ambiguous.

For example, consider two phases of one experiment, one called "training" and the other called "testing," in which an agent is first trained and then evaluated in another environment configuration. In arsenAI, there are three ways to define the second phase: (1) Not mentioning the agent and using the cascading configuration feature of arsenAI; (2) using a YAML anchor, or (3) simply copying the definition of the previous phase.

Thus, a hash value needs to be computed of an experiment document in order to allow unique identification of documents. To compute a hash value, all idiosyncrasies must be removed and the resulting document then hashed. For this, the experiment document must first be fully expanded, i. e., the cascading configuration explicitly spelled out (cf. Figure 4. Afterwards, we convert the YAML document into JSON in the most compact form possible, since the YAML format allows—by design—for a number of ambiguities. Finally, we hash the resulting character stream. The conversation is easily possible because YAML is a superset of JSON [18]; thus, YAML documents can always be reduced to JSON. The stricter syntax of JSON ensures the deterministic, non-ambiguous minimal form (i. e., with all unnecessary whitespace removed), if only commas and colons are accepted as separators and keys to objects are lexicographically sorted. Hashing the result of `to_json(expanded_experiment_document, separators=(',', ':'), sort_keys=True)` will be unambiguous across different machines.

IV. Evaluating Results

Usually, results evaluation means reporting created by a human for a particular experiment, such as a Jupyter notebook with custom-made plots or other means of (statistical) analysis. While this is the usual way for an analysis, e. g., prior to publication, it most commonly implies manual verification of numbers and plotted graphs. However, when learning agent systems are an integral part of a co-simulated CPES experiment, there are many metrics to consider, as well as their interdepedence and the factors the influence them.

Usually, presentations begin with the graph of the reward or utility function, plotted by time or episode. For the example given, Figure 5 depicts an initial plot and analysis of the agent's utility functions that seem to indicate that the autocurriculum-training does indeed lead to a better agent policy. However, additional analysis is warranted, giving a box or violin plot. Also, key metrics from the environment are presented, such as voltage magnitudes in the case of power grid simulations. As the reward/utility function of an agent is a user-defined piece of code, the correlation between observed values and the reward should be inspected. Moreover, an agent's apparent success could be due to an advantageous initial state, so these calculations should be performed given several experiment runs with different random seeds. If time series data (e. g., weather data) are part of the simulation, the starting date must be varied, too. For a sound analysis of the experiment, there are usually more metrics that can be considered, depending on the actual subject of the investigation. For DRL agents, the entropy value during training is often of interest to investigate the exploration-exploitation trade-off.

Most often, these particular values under investigation have a close relationship to each other; whether an experiment is supporting the formulated hypothesis or not then hinges on thresholds either for individual values or for their correlation factor. I. e., an experimenter can—or rather, should—usually formulate beforehand whether a hypothesis is validated or refuted. This can be done through invariants, or, more broadly, user-supplied code that, in addition to the usual analysis, returns a boolean value that indicates the state of the hypothesis.

palaestrAI stores results in a database and provides a convenience query interface that allows to quickly retrieve the most common values from it. This way, custom results validation functions can be provided as part of an experiment. The `invariants` key of an experiment (run) definition document maps to a list of names that reference classes. At the end of an experiment run, each is instantiated and their `check()` method called. This method receives the reference to the current experiment run's data in the results database. If all check functions return `true`, the hypothesis is validated.

The "experimentation pipeline" described allows not only to create a hypothesis validator for a particular experiment, but also to provide numerous building blocks for hypothesis validation that can be used in a number of experiment. For example, there are definite boundaries for voltage magnitudes. One particular check, called `VoltageHealth`, would then check that no bus would ever see a voltage magnitude outside of the range $0.85 \leq V \leq 1.15\,\text{p.u.}$

V. Conclusion and Future Work

In this paper, we have presented a way to conduct sound and reliable experimentation with learning agents in co-simulated CPES. This includes the presentation of a experiment definition document, as well as motivating the most important design as aspects of this approach.

The approach of the experiment definition document as well as the tool suite offer a number of benefits. First, the format is both easy for humans to write, yet also easy to process by the framework. This notion not just includes the text format (YAML), but also its structure, which allows easy instantiation of software components on-the-fly to configure the simulation. Together with arsenAI's DoE features, even large studies can be formulated in one file and the executed also at large scale, with palaestrAI allowing to utilize multiple Graphics Processing Units (GPUs) as well as a fleet of containers.

However, the instantiation also gives rise to a practical challenge. Software modules are created at the start of the simulation, their parameters are—depending in the particular developer's coding style—evaluated when first used. Thus, errors in the code are caught at a later point, compared to a monolithic setup or other approach that allows static analysis. Therefore, finding and fixing bugs in one's experiment setup can be prone to longer startup cycles. Furthermore, even though basic syntax checking through YAML schema exists, there is no

 ISBN: 978-1-68558-139-8

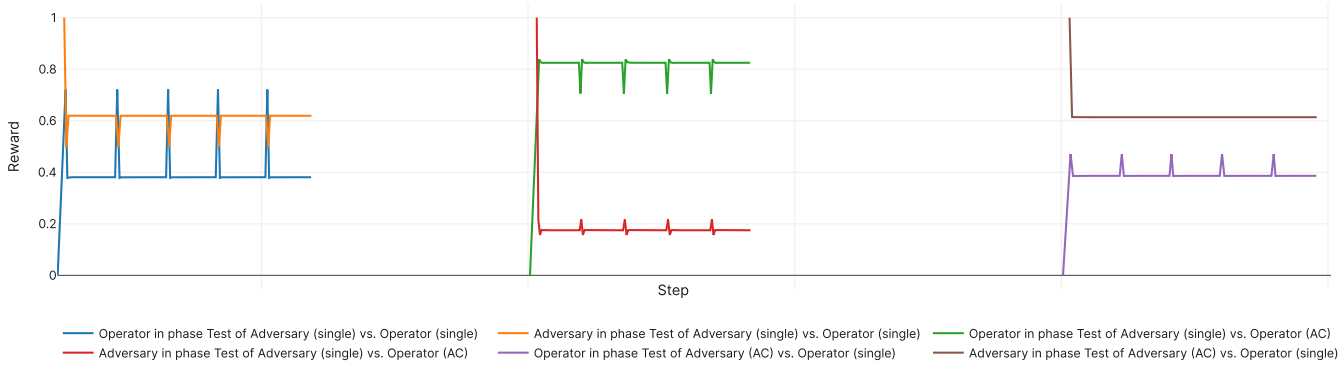

Figure 5. Preliminary results from the autocurriculum experiment (cf. Figures 2 and 3)

semantic help or specific syntax highlighting as fully integrated simulation environments can provide.

In the future, we will showcase an extensive experiment that investigates the benefits of autocurriculum learning with the ARL methodology and demonstrate how the particular elements of our approach tie into a validation of a scientific hypothesis in a largely automated fashion, driven by an experiment definition document and the palaestrAI software suite.

Acknowledgements

This work was funded by the German Federal Ministry for Education and Research (BMBF) under Grant No. 01IS22071.

References

[1] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] J. Schrittwieser *et al.*, *Mastering Atari, Go, Chess and Shogi by planning with a learned model*, 2019. arXiv: 1911.08265.

[3] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019, Number: 7782 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://www.nature.com/articles/s41586-019-1724-z (visited on 01/29/2024).

[4] E. M. Veith, *Universal Smart Grid Agent for Distributed Power Generation Management*. Logos Verlag Berlin GmbH, 2017.

[5] R. Diao *et al.*, "Autonomous voltage control for grid operation using deep reinforcement learning," *IEEE Power and Energy Society General Meeting*, vol. 2019-Augus, 2019, ISSN: 19449933. DOI: 10.1109/PESGM40551.2019.8973924. arXiv: 1904.10597.

[6] B. L. Thayer and T. J. Overbye, "Deep reinforcement learning for electric transmission voltage control," in *2020 IEEE Electric Power and Energy Conference (EPEC)*, IEEE, 2020, pp. 1–8.

[7] Z. Wu, C. Li, and L. He, "A novel reinforcement learning method for the plan of generator start-up after blackout," *Electric Power Systems Research*, vol. 228, p. 110 068, Mar. 1, 2024, ISSN: 0378-7796. DOI: 10.1016/j.epsr.2023.110068. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378779623009550 (visited on 01/29/2024).

[8] K. Arshad *et al.*, "Deep reinforcement learning for anomaly detection: A systematic review," *IEEE Access*, vol. 10, pp. 124 017–124 035, 2022. DOI: 10.1109/ACCESS.2022.3224023.

[9] E. Veith, A. Wellßow, and M. Uslar, "Learning new attack vectors from misuse cases with deep reinforcement learning," *Frontiers in Energy Research*, 2023.

[10] T. Wolgast *et al.*, "Analyse–learning to attack cyber-physical energy systems with intelligent agents," *SoftwareX*, Apr. 2023. DOI: 10.1016/j.softx.2023.101484. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2023SoftX..2301484W/abstract.

[11] R. A. Fisher, "Design of experiments," *British Medical Journal*, vol. 1, no. 3923, p. 554, Mar. 14, 1936, ISSN: 0007-1447. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2458144/ (visited on 01/29/2024).

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

[13] H. Neema, J. Sztipanovits, M. Burns, and E. Griffor, "C2wt-te: A model-based open platform for integrated simulations of transactive smart grids," 2016. DOI: 10.1109/MSCPES.2016.7480218.

[14] H. Neema *et al.*, "Simulation integration platforms for cyber-physical systems," in *Proceedings of the Workshop on Design Automation for CPS and IoT*, ser. DESTION '19, New York, NY, USA: Association for Computing Machinery, Apr. 15, 2019, pp. 10–19, ISBN: 978-1-4503-6699-1. DOI: 10.1145/3313151.3313169. [Online]. Available: https://dl.acm.org/doi/10.1145/3313151.3313169 (visited on 01/28/2024).

[15] S. Balduin, E. M. S. P. Veith, and S. Lehnhoff, "Midas: An open-source framework for simulation-based analysis of energy systems," in *Simulation and Modeling Methodologies, Technologies and Applications*, G. Wagner, F. Werner, and F. De Rango, Eds., Cham: Springer International Publishing, 2023, pp. 177–194, ISBN: 978-3-031-43824-0.

[16] E. Veith *et al.*, "palaestrAI: A training ground for autonomous agents," in *Proceedings of the 37th annual European Simulation and Modelling Conference*, EUROSIS, Oct. 2023.

[17] M. Hutson, "Artificial intelligence faces reproducibility crisis," *Science*, vol. 359, no. 6377, pp. 725–726, Feb. 16, 2018, Publisher: American Association for the Advancement of Science. DOI: 10.1126/science.359.6377.725. [Online]. Available: https://www.science.org/doi/full/10.1126/science.359.6377.725 (visited on 01/29/2024).

[18] O. Ben-Kiki, C. Evans, and I. döt Net, *YAML ain't markup language (YAML™) version 1.2*, [Retrieved: 2024-02-09]. [Online]. Available: https://yaml.org/spec/1.2.2/#102-json-schema.