Modeling Machine Learning Concerns in Collective Adaptive Systems

Petr Hnětynka[a], Martin Kruliš[b], Michal Töpfer[c] and Tomáš Bureš[d]
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

Keywords: Collective Adaptive Systems, Machine Learning, Model-Driven, Meta-Model.

Abstract: Collective adaptive systems (CAS) are systems composed of a large number of heterogeneous entities without central control that adapt their behavior to reach a common goal. Adaptation and collaboration in such systems are traditionally specified via a set of logical rules. Nevertheless, such rules are often too rigid and do not allow for the evolution of a system. Thus, recent approaches started with the introduction of machine learning (ML) methods into CAS. In the is paper, we present a model-driven approach showing how CAS, which employs ML methods for adaptation, can be modeled—on both the platform independent and specific levels. In particular, we define a meta-model for modeling CAS and a mapping of concepts defined in the meta-model to the Python framework.

1 INTRODUCTION

Collective adaptive systems (CAS) (Anderson et al., 2013) are commonly understood as systems composed of a large number of heterogeneous entities without central control. The entities adapt their behavior in order to reach a collective goal. Contemporary smart systems like smart homes, smart cities, smart agriculture, or Industry 4.0 management systems typically fall into the category of CAS.

Adaptation and collaboration in such systems are traditionally specified via a set of rules (i.e., logical hard and soft constraints) that are continuously evaluated. While the specification via the rules is straightforward and easy to understand, they are often too rigid and do not allow for the evolution of the system when its context gradually changes.

Therefore, many recent approaches (Muccini and Vaidhyanathan, 2019; Gheibi et al., 2021a; Saputri and Lee, 2020; Weyns et al., 2021) have started with the introduction of machine learning (ML) methods, especially neural networks, to the adaptation loop. Usually, ML methods in these approaches are added in an ad-hoc manner, are “hidden” in their implementation, and there is a lack of support on the architectural level for a system components specification that provides abstractions for ML. Thus, we have introduced the concepts of *estimators* (Töpfer et al., 2022), which are architectural-level objects providing predictions about a particular quantity, and which are internally backed by ML methods.

In this paper, we formalize the concept of the estimators using meta-models and show how the models are transformed to specification in Python. The formalization of estimators represents a platform-independent model while the mapping to Python is a particular platform-specific one.

The presented approach is described in the scope of the DEECo ensemble-based component model (Bureš et al., 2020), but it is applicable for CAS that are modeled using components.

The work in this paper is a continuation of our work presented in (Bureš et al., 2022), where we have designed a model-driven approach of a gradual transformation of system models that describe their behavior via logical rules into models with behavior specified via specially constructed neural networks while keeping a clear relation to the original logical rules. With the introduction of estimators, the logical rules can be directly enhanced with ML capabilities without a need for extra steps in design and development.

The main contribution of this paper is a complete model-driven approach for collective adaptive systems which employ ML methods. Our approach makes it possible to model such systems and the ML inference on a platform-independent level (via a set of meta-models) and on platform-specific levels (via a mapping to Python).

The paper is structured as follows. Section 2 describes a running example and the basics of ensemble-based component models, including the DEECo model.

[a] https://orcid.org/0000-0002-1008-6886
[b] https://orcid.org/0000-0002-0985-8949
[c] https://orcid.org/0000-0002-3313-1766
[d] https://orcid.org/0000-0003-3622-9918

Hnětynka, P., Kruliš, M., Töpfer, M. and Bureš, T.
Modeling Machine Learning Concerns in Collective Adaptive Systems.
DOI: 10.5220/0011693300003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 55-62
ISBN: 978-989-758-633-0; ISSN: 2184-4348
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

Section 3 formalizes the concept of estimators via the meta-model, and Section 4 describes its mapping to the Python constructs. Section 5 discusses related work and Section 6 concludes the paper.

2 RUNNING EXAMPLE AND BACKGROUND

Our running example is a simplified use-case taken from our recently successfully finished ECSEL JU project AFarCloud[1].

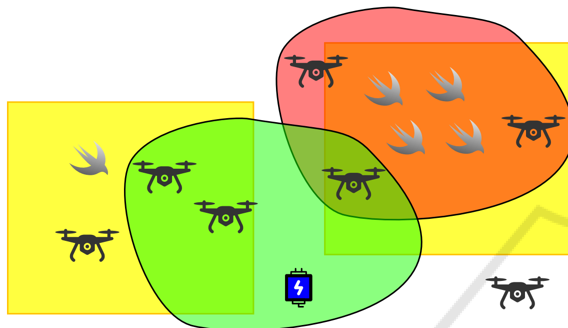

Figure 1: Running example.

In the example (that is visualized in Figure 1), there is a farm with a number of fields with crops (yellow rectangles) that need protection from flocks of birds. The farm is monitored by a number of drones that observe the fields and perform various measurements. Also, drones are employed to scare flocks away from the fields (to farm areas that are not "bird-sensitive"). Typically, a group of drones is necessary to scare a flock away from the fields. Additionally, the drones operate with a limited battery capacity and need to recharge at the charger (the blue square with the lightning icon in the figure), which can charge only a limited number of drones at a single time.

It is obvious that drones need to cooperate closely and adapt their behavior in order to work successfully. First, the drones need to form groups to scare flocks away, and the size and position of the group depend on the size and position of the flock. Second, the drones need to cooperate in order to optimize their battery charge level and charger utilization, ensuring drones do not terminate without energy and the charger is used as much as possible.

2.1 Ensembles

To model such adaptive systems, we are exploiting the power of *autonomic component ensembles*. Namely, we are using the DEECo ensemble-based component model (Bureš et al., 2020).

Figure 2 shows a meta-model (simplified to save space) describing the core concepts of the ensembles.

Using DEECo, entities in a system are modeled as Components that are instances of the ComponentType meta-class. ComponentType declares component fields (i.e., data) that contain a state of a particular component. Also, ComponentType declares component actions, i.e., its behavior. In the case of the example, there are three component types: (i) for the drones, (ii) for the charger, and even (iii) for the flocks. In the case of the flocks, the components are "beyond direct control", and thus their state can be observed only.

Ensembles are dynamically established groups of components and model interactions among the components, and they are instances of EnsembleType. Formally, the ensemble type definition consists of the following: (i) a priority providing ordering, in which the ensembles are evaluated; (ii) an Action to be performed on components grouped in the ensemble; and (iii) a set of Roles that determine which components are to be included in the ensemble. A role is either StaticRole or DynamicRole. Both kinds of roles define cardinality, i.e., how many components of the same type have to be set for the role. The static roles are specified when the ensemble is instantiated and cannot change. The dynamic ones are populated in an adaptive way, given the situation in the system. Technically, the dynamic roles have a Selector consisting of a predication determining if a component may be selected for the particular role, and the utility function (which orders the components selected for the role, if there are more potential components than is its cardinality).

Figure 3 shows a part of the Drone component type that is instantiated by individual drones. It has the battery energy field, the drone state field, and others (omitted due to space constraints) and actions to fly to a position, return to the charger, etc. In a similar manner, other components (Charger, Field, Flock, etc.) are defined.

In the running example, there are two ensemble types: the DroneChargingAssignment ensemble type grouping drones with a charger (the green group in Figure 1), and the FieldProtections ensemble type grouping drones to protect a particular field (the red group in Figure 1). The former ensemble type is instantiated once per each Charger component; the latter one is instantiated once per a field in danger.

An excerpt of DroneChargingAssignment is depicted in Figure 4. A particular charger, for which the ensemble is instantiated, is assigned to the ensemble static role charger and cannot be later reassigned. Drones to the ensemble dynamic role are selected ac-

[1]https://www.ecsel.eu/projects/afarcloud

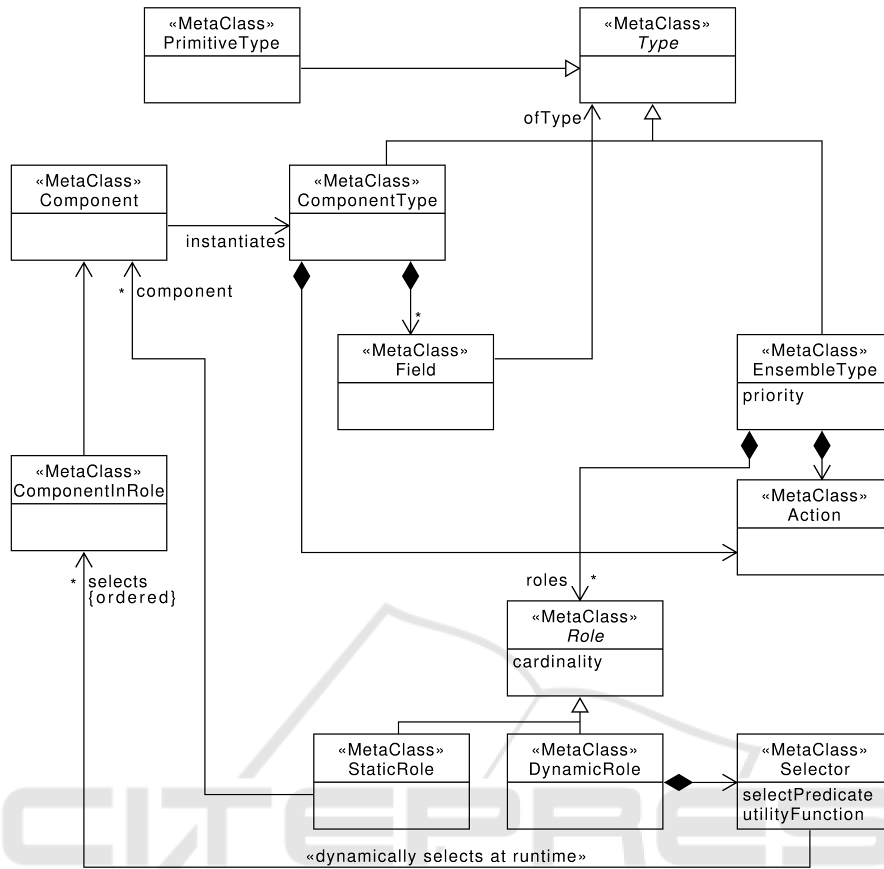

Figure 2: Meta-model of ensemble-based component architectures.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

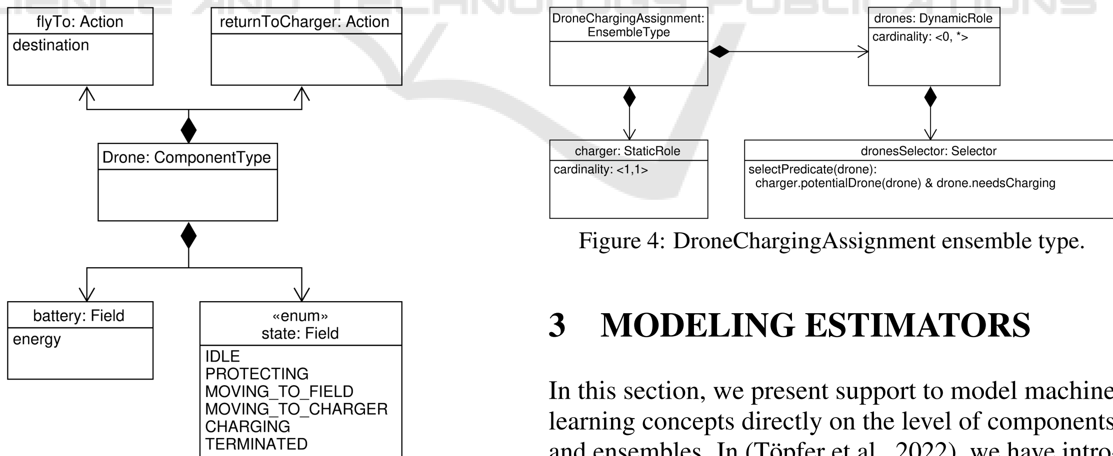

Figure 3: Drone component type.

cording to the role Selector, which select drones in-need-to-be-charged, which are close to the particular charger. As the cardinality is unlimited, there is no utility function defined (as all the drones can be assigned to the role).

The FieldProtection ensemble is defined similarly.

Figure 4: DroneChargingAssignment ensemble type.

3 MODELING ESTIMATORS

In this section, we present support to model machine learning concepts directly on the level of components and ensembles. In (Töpfer et al., 2022), we have introduced the concepts of *estimators*. An estimator is an object, which provides predictions about a particular quantity. We limit the scope of this work to supervised ML only. This allows for predicting quantities, for which the true value will be available later during the run of the system, e.g. predicting a future state of a component.

The estimator concept itself is a general one and can be used in any approach for modeling collective

adaptive systems. In ensemble-based modeling, it can be attached to a component, ensemble, or pair component-ensemble. In either case, there can be multiple estimators attached to a given entity, and each of the estimates predicts another value. Regarding the predicted value, estimators can be divided to be used for:

(i) classification—predicting a value from a predefined fixed set of possibilities, e.g., possible states of a component,

(ii) regression—predicting a continuous value, and

(iii) time-to-condition—predicting when a condition becomes true.

Internally, each estimator is implemented by a machine learning model (e.g., neural network) that performs predictions. Each estimator can have a number of inputs via which data are collected for the training of the internal ML model.

In the running example, the Drone component has two estimators: one predicting how long it will take for the drone to start charging, and another one predicting its battery state at time instant in the future. These estimators are then used in the drone to decide when to start to signal the drone needs to be charged. Similarly, the DroneChargingAssignment ensemble type defines a component-ensemble estimator used in the ensemble selector to select the best drone for charging.

In the rest of the section, we present the estimators meta-model in detail, together with particular examples of usage in the running example.

3.1 Estimators Meta-Model

Figure 5 shows the meta-model of the above-discussed estimators and how they are incorporated into DEECo (technically, this meta-model is a package that extends the core meta-model in Figure 2—the gray dashed elements are defined in the core meta-model).

The core element is the Estimate, which represents values to be learned together with all the necessary inputs, guards, etc. The Estimate can be attached to a component (each component can have multiple Estimates—each for a different data field), an ensemble, or a pair ensemble-component.

The Estimate itself is parameterized by the EstimatorModel, which defines parameters for the underlying neural network and thus the estimate implementation and behavior. Each Estimate can have multiple Inputs (training features), i.e., fields of the component needed for training and prediction. We distinguish here between numerical and categorical features, which influences whether the value is used as-is (possibly normalized) or whether one-hot encoding (in the case of categorical features) is used.

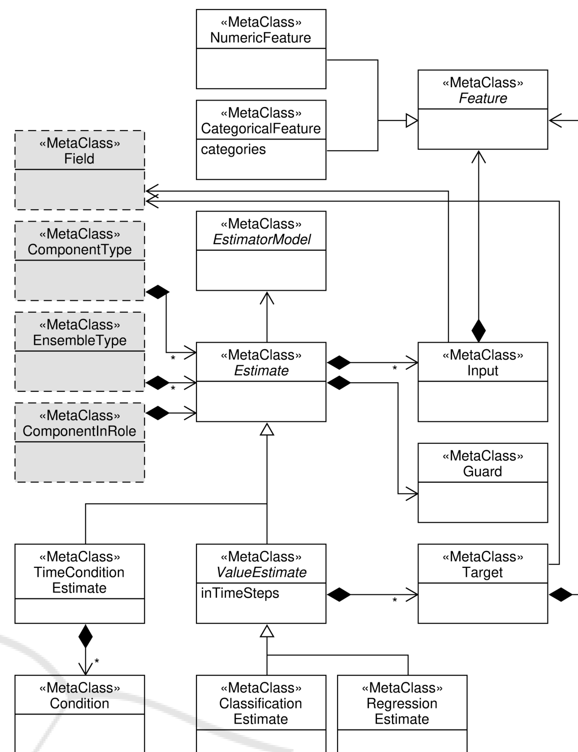

Figure 5: Estimate meta-model.

The Estimate is further specialized to distinguish between the options “what” it predicts. In the *value* case (represented by subclass ValueEstimate), it specifies a target, which denotes the truth values that are to be predicted by the estimator. This can be either a numerical or a categorical value computed based on the component fields. For numerical values, we use the RegressionEstimate subclass of ValueEstimate, and for categorical values, we use the ClassificationEstimate. The number of time steps we want to predict into the future is set by the inTimeSteps attribute of ValueEstimate.

For the *time-to-condition* case, there is another subclass of Estimate—TimeToConditionEstimate—which specifies a required condition.

The Estimate further defines a guard predicate (over component fields), which determines if inputs and outputs (i.e., the target feature or the result of the condition) are valid and thus can be used to collect data for training the estimator.

Such a description of an estimator is enough for automated data collection and training. As already stated, we focus on supervised ML tasks, so we assume the correct values for the estimator predictions will be observed later during the run of the simulation. The semantics of the modeling concepts in the data collection phase is as follows.

In the case of the ValueEstimate, we perform the following actions in every time step:

1. We collect the inputs and the current time provided that the guard condition on inputs is true.
2. We collect the true outputs (represented by class Target) for the predictions realized earlier during the run of the system, provided that the guard condition on the output is true. We associate the output with inputs that were collected inTimeSteps time steps ago. If the guard condition on the output is false, we discard the inputs recorded inTimeSteps time steps ago.

In the case of the TimeToConditionEstimate, we perform the following action in every time step:

1. We collect the inputs and the current time to a buffer provided that the guard condition on inputs is true.
2. If the condition specified by the Condition is true, we associate all the inputs collected in the buffer (as per step #1) with the difference between the current time and the time of the input in the buffer. We clear the buffer.

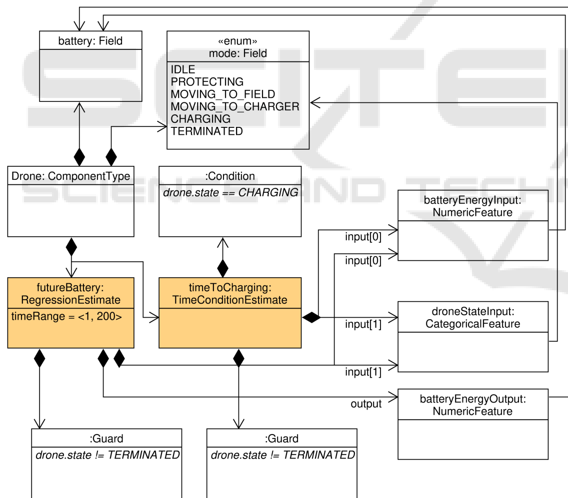

Figure 6: Drone component.

To illustrate the concepts, Figure 6 shows an instance of the meta-model for the drone component of the running example. It extends the model from Figure 3. In addition to the fields and actions, the drone has attached two estimates. The first one—TimeToConditionEstimate—predicts how long it will take for the drone to get into the CHARGING state (thus the Condition is a simple predicate checking equality of the State to the CHARGING value). The estimate has two inputs—BatteryEnergyInput and DroneStateInput (the former of the numeric kind while the latter of the categorical kind). Similarly, the second one—FutureBatteryEstimate—is for predicting the battery energy.

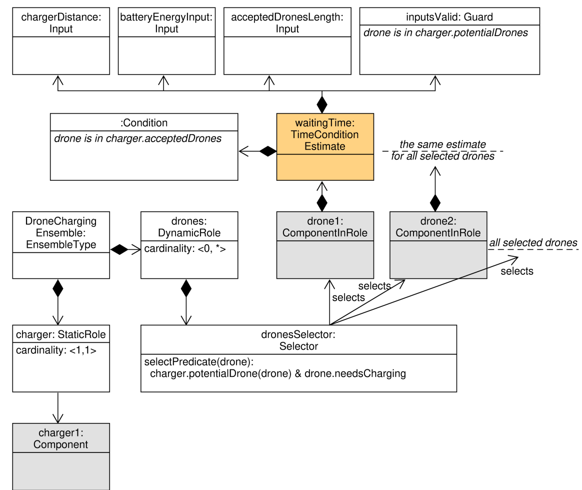

Figure 7: DroneChargingAssignment ensemble.

Figure 7 shows an example of the component-ensemble attached estimator. It is applied to components selected in the DroneChargingAssignment ensemble (the model extends the model from Figure 4). The estimate predicts time how long the drone will be waiting for the charger. As inputs, the estimate takes the distance to the charger, the drone battery energy level, and the number of drones waiting for the charger. All the inputs are valid only when the drone is considered for the charger as defined by the guard condition.

4 MAPPING TO PYTHON

As an evaluation and proof of concept, we developed an open-source Python-based framework that realizes the approach described in Section 3. The Python framework represents a particular platform-specific model to which the concepts defined in the previous section are mapped. The framework features API for defining components, ensembles, and estimators—thus providing an internal domain-specific language for the design of ensemble-based component systems that employ machine learning.

The framework uses decorators[2] to define inputs, expected outputs (value or condition), and guards for the estimators—exactly following the concepts in the meta-model shown in Figure 5.

[2]A decorator in Python is a function/method that takes the function/method over which is applied and assigns the result to the identifier of the original function/method—see https://www.python.org/dev/peps/pep-0318/.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Both the component and ensemble types are defined as classes.

Listing 1 shows the definition of the component type Drone. The class extends the predefined Component class. The fields of the component are defined in the constructor (i.e., the __init__ method).

The Drone component showcases two estimators as defined in the model in Figure 6—one for battery level estimation and another for estimating the time till charging is needed. The battery level estimator uses the current battery level and the state of the drone as inputs and predicts the battery level 50 time steps in the future. The definition of the estimator is split into three parts: (a) The definition of the machine learning model and storage for the collected data (lines 1–4). This part is specific to the implementation, thus we do not reflect it in the meta-model. (b) The declaration of the estimate field in the component; this corresponds to the association from the ComponentType to Estimate in the meta-models (lines 8–9). (c) The definition of inputs, targets, and guards. These are realized as decorators on component fields and getter functions of the component.

Namely, the decorators are as follows. The @futureBatteryEstimate.input() decorates methods returning input values. The inputs are marked whether they are numeric values or categorical ones. The target is similarly decorated with the @futureBatteryEstimate.target() (line 33). The @futureBatteryEstimate.inputsValid() and @futureBatteryEstimate.targetsValid() (starting at line 42) denote guards, i.e., conditions under which the inputs and targets can be used for training the estimators (in this particular case, the drone must not be in the TERMINATED state—line 46).

The definition of the estimator for the time till charging is needed is very similar. The only difference is that instead of defining the target, a condition is provided (line 48).

```
1  droneBatteryEstimator = NeuralNetworkEstimator(
2      hidden_layers=[32, 32], # two hidden layers with 32
           neurons
3      name="Drone battery"
4  )
5  timeToChargingEstimator = ...
6
7  class Drone(Component):
8      futureBatteryEstimate =
           ValueEstimate().inTimeSteps(50)\
9                      .using(droneBatteryEstimator)
10     timeToChargingStateEstimate = TimeEstimate()\
11                     .using(timeToChargingEstimator)
12
13     def __init__(self, location):
14         self.battery = 1
15         self.state = DroneState.IDLE
16         # more code
17
18     @futureBatteryEstimate.\
19             input(NumericFeature(0, 1))
20     @timeToChargingStateEstimate.\
21             input(NumericFeature(0, 1))
22     def battery(self):
23         return self.battery
24
25     @futureBatteryEstimate.\
26         input(CategoricalFeature(DroneState))
27     @timeToChargingStateEstimate.input(
28         CategoricalFeature(DroneState))
29     def drone_state(self):
30         return self.state
31
32     @futureBatteryEstimate.\
33              target(NumericFeature(0, 1))
34     def battery(self):
35         return self.battery
36
37     @timeToChargingStateEstimate.target(
38         CategoricalFeature(DroneState))
39     def drone_state(self):
40         return self.state
41
42     @futureBatteryEstimate.inputsValid
43     @futureBatteryEstimate.targetsValid
44     @timeToChargingStateEstimate.inputsValid
45     def not_terminated(self):
46         return self.state != DroneState.TERMINATED
47
48     @timeToChargingStateEstimate.condition
49     def is_charging_state(self):
50         return self.state == DroneState.CHARGING
51
52     def actuate(self):
53         # more code
```

Listing 1: Drone component specification.

Ensemble types are specified in a way similar to components. This is illustrated in Listing 2, which shows the DroneChargingAssignment ensemble type that we already introduced in Section 2 and its estimators described Section 3.

The static role (charger) is declared on line 8. The value of the role is set in the constructor of the ensemble.

The dynamic role (drones) is declared on line 11. Its content is set dynamically by the framework based on the actual state, position, and battery levels of the drones. The declaration of the role is done via the someOf function, which denotes that the role is a collection. (An opposite would be oneOf, which would mean the role contains only one instance.) The declaration of the drones role further contains the declaration of the estimator that is associated with the role. This we will describe later in the text.

Cardinality and selector of the dynamic role are declared using decorated methods of the ensemble. In this case, the cardinality is defined on line 13. The cardinality is a tuple containing the lower and the upper bound. The selector is defined on line 17.

The ensemble priority (influencing order in which

```
1  class DroneChargingAssignment(Ensemble):
2      def __init__(self, charger: 'Charger'):
3          self.charger = charger
4
5      def priority(self): return 2
6
7      # static role
8      charger: Charger
9
10     # dynamic role
11     drones: List[Drone] =
           someOf(Drone).withTimeEstimate()\
12                 .using(waitingTimeEstimator)
13     @drones.cardinality
14     def drones(self):
15         return 0, ENVIRONMENT.droneCount
16
17     @drones.select
18     def drones(self, drone, otherEnsembles):
19         waitingTime = self.drones.estimate(drone)
20         return drone in self.charger.potentialDrones and
               drone.needsCharging(waitingTime)
21
22     @drones.estimate.input(NumericFeature(0, 1))
23     def battery(self, drone):
24         return drone.battery
25
26     @drones.estimate.input(NumericFeature(0,
           ENVIRONMENT.size))
27     def charger_distance(self, drone):
28         return self.charger.location.\
29                 distance(drone.location)
30
31     @drones.estimate.
32         input(NumericFeature(0,
               ENVIRONMENT.chargerCapacity))
33     def accepted_drones_length(self, drone):
34         return len(self.charger.acceptedDrones)
35
36     @drones.estimate.inputsValid
37     def is_preassigned(self, drone):
38         return drone in self.charger.potentialDrones
39
40     @drones.estimate.condition
41     def is_accepted(self, drone):
42         return drone in self.charger.acceptedDrones
43
44     def actuate(self):
45         self.charger.waitingDrones = self.drones
```

Listing 2: Ensemble specification.

the ensembles are instantiated) is specified as the method priority (line 5).

The action that gets periodically executed by the ensemble for the member components is given in the function actuate (on lines 44—45).

The DroneChargingAssignment ensemble type illustrates the most complex case of estimation when the estimator is associated dynamically with a component in the context of the ensemble. In this case, it is the TimeToConditionEstimate predicting the waiting time before an actual charging of the drone can start (this includes the time needed to fly to the charger and the time a drone has to circle around the charger to be given a free slot in which it can charge).

The configuration of the estimator is similar to the example in Listing 1. The specification of the estimator is split into three parts: (a) The definition of the machine learning model—this is exactly the same as already explained. (b) The declaration of the estimator in the ensemble differs in that we are associating the estimator with the ensemble-component pair. Following the meta-model in Figure 5, the estimator is associated with a role in the ensemble, which stems from the transitive relationship Role→ComponentInRole→Estimate. This is reflected in the code using the withTimeEstimate method used in role declaration (line 11). (c) The definition of inputs and the target condition is again the same as before, only associated with the ensemble role. Namely, the inputs are specified on lines 22–34, the target condition is given on lines 40–42 and the guard is given on lines 36–38.

5 RELATED WORK

Related approaches to our one are any that employ machine-learning techniques to collective adaptive systems. As stated already in the introduction, there are a number of such approaches, but most of them incorporate ML techniques "under the hood" and "hard-coded" in implementations. Typically, ML is employed in the planning and adaptation phases of the adaptation loop as it is confirmed by several systematic literature reviews (e.g., (Saputri and Lee, 2020; Gheibi et al., 2021a)). Below, we discuss several particular approaches that are the closest ones.

A number of approach use ML methods to reduce the size of an adaption space. These are for example (Van Der Donckt et al., 2020; Van Der Donckt et al., 2020; Gheibi et al., 2021b). In (Cámara et al., 2020), an approach combining machine learning and probabilistic model checking is described, which again tries to select the best possible adaptations and thus achieve optimal decisions. In (Muccini and Vaidhyanathan, 2019), ML methods are used in both the monitoring and analysis phases of the MAPE-K loop to forecast future values of QoS parameters of a given system and therefore to select an optimal adaptation strategy. As mentioned, these approaches use ML internally and, contrary to our approach, do not bring ML capabilities to the level of architectural specifications of systems.

A large but not closely related usage of ML methods in adaptive systems is for attack and anomaly detection (approaches overviewed e.g., in (Mohammadi Rouzbahani et al., 2020)).

Regarding the explicit modeling of adaptive systems and model-driven approaches for their develop-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

ment, approaches can be found in (D'Angelo et al., 2018) or (Weyns and Iftikhar, 2019). Nevertheless, they do not integrate ML techniques.

6 CONCLUSION

In the paper, we have presented an approach for the formal modeling of machine learning concepts in collective adaptive systems. We have presented the concept of estimators, which are architectural-level objects providing predictions about a particular quantity, and defined a meta-model for them. Also, we have proposed a mapping of concepts defined by the meta-model to the Python framework, which thus represents a particular platform-specific model.

While the Python framework is fully functional, our future work is twofold. Currently, we are working on incorporating additional machine learning methods to be used as estimators implementation. Additionally, we plan to provide an automated transformation from the platform-independent specifications to the Python framework.

ACKNOWLEDGMENTS

This work has been partially supported by the Czech Science Foundation project 20-24814J and also partially supported by Charles University institutional funding SVV 260588.

REFERENCES

Anderson, S., Bredeche, N., Eiben, A., Kampis, G., and van Steen, M. (2013). *Adaptive collective systems: Herding black sheep*. Bookprints.

Bureš, T., Gerostathopoulos, I., Hnětynka, P., Plášil, F., Krijt, F., Vinárek, J., and Kofroň, J. (2020). A language and framework for dynamic component ensembles in smart systems. *International Journal on Software Tools for Technology Transfer*, 22(4):497–509.

Bureš, T., Hnětynka, P., Kruliš, M., and Pacovský, J. (2022). Towards model-driven fuzzification of adaptive systems specification. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development*, pages 336–343.

Cámara, J., Muccini, H., and Vaidhyanathan, K. (2020). Quantitative Verification-Aided Machine Learning: A Tandem Approach for Architecting Self-Adaptive IoT Systems. In *Proceedings of ICSA 2021, Salvador, Brazil*, pages 11–22. IEEE.

D'Angelo, M., Napolitano, A., and Caporuscio, M. (2018). CyPhEF: a model-driven engineering framework for self-adaptive cyber-physical systems. In *Companion Proceedings of ICSE 2018, Gothenburg, Sweden*, pages 101–104. ACM.

Gheibi, O., Weyns, D., and Quin, F. (2021a). Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review. *ACM Transactions on Autonomous and Adaptive Systems*, 15(3):9:1–9:37.

Gheibi, O., Weyns, D., and Quin, F. (2021b). On the Impact of Applying Machine Learning in the Decision-Making of Self-Adaptive Systems. In *Proceedings of SEAMS 2021, Madrid, Spain*, pages 104–110. IEEE.

Mohammadi Rouzbahani, H., Karimipour, H., Rahimnejad, A., Dehghantanha, A., and Srivastava, G. (2020). Anomaly Detection in Cyber-Physical Systems Using Machine Learning. In *Handbook of Big Data Privacy*.

Muccini, H. and Vaidhyanathan, K. (2019). A machine learning-driven approach for proactive decision making in adaptive architectures. In *Companion Proceedings of ICSA 2019, Hamburg, Germany*, pages 242–245.

Saputri, T. R. D. and Lee, S.-W. (2020). The Application of Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *IEEE Access*, 8:205948–205967.

Töpfer, M., Abdullah, M., Bureš, T., Hnětynka, P., and Kruliš, M. (2022). Ensemble-based modeling abstractions for modern self-optimizing systems. In *Proceedings of ISOLA 2022, Rhodes, Greece*, volume 13703 of *LNCS*, pages 318–334. Springer.

Van Der Donckt, J., Weyns, D., Quin, F., Van Der Donckt, J., and Michiels, S. (2020). Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In *Proceedings of SEAMS 2020, Seoul, South Korea*, pages 20–30. ACM.

Weyns, D. and Iftikhar, M. U. (2019). ActivFORMS: A Model-Based Approach to Engineer Self-Adaptive Systems. *arXiv:1908.11179 [cs]*.

Weyns, D., Schmerl, B., Kishida, M., Leva, A., Litoiu, M., Ozay, N., Paterson, C., and Tei, K. (2021). Towards Better Adaptive Systems by Combining MAPE, Control Theory, and Machine Learning. In *Proceedings of SEAMS 2021, Madrid, Spain*, pages 217–223. IEEE.