Towards XCSF-based Identification of Physical Disturbances

Markus Görlich-Bucher[1], Jörg Hähner[1]

[1] *Organic Computing Group, University of Augsburg, Germany*

Abstract
Building robust systems that are able to withstand various kinds of disturbances is an ongoing subject of research in the broader field of intelligent systems. Hereby, *physical disturbances*, therefore, disturbances that affect a system's hardware, are rarely discussed. We present an XCSF-based approach for identifying physical disturbances. We utilize a concept for limiting XCSF's necessary supervision as well as various system metrics in form of *triggers* in order to reduce the amount of interaction with human experts. We evaluate our approach using a simple proof of concept, showing that an XCSF with our extensions is able to perform acceptable compared to a naive XCSF. Besides, we discuss possible future research direction for our approach.

1. Introduction

The increasing complexity in Information and Communication Technology has led to the emergence of various research initiatives concerned with building intelligent systems that incorporate several so-called self-x capabilities, such as *Organic Computing* (OC) ([1]), *Autonomic Computing* ([2]), or, most recently, *Lifelike Computing* ([3]). These initiatives are concerned with building robust and flexible systems that are able to interact with their environment using sensors and actuators. They should be able to freely adapt and organize themselves in order to pursue their goals. A well-discussed characteristic of such systems is *Robustness*: The ability to withstand internal or external *disturbances* and remain in (or return to) a desired state of function.

As suggested in [4], a yet rarely considered aspect of disturbances lies in disturbances that are hardware related, for example due to degrading or breaking actuators. Such disturbances are associated with various drawbacks: First of all, they might not necessarily get noticed by the containing system. If at all, the system is able to retrieve some internal measurements from the broken component, which does not directly provide an assessment of its state at all. Hence, it is of interest to implement some sort of learning mechanism that iteratively learns to identify broken components. However, providing some sort of necessary ground truth for learning intuitively involves some sort of human expert that is able to asses the physical state of the system. This results in a second drawback: Human experts are usually associated with some sorts of costs, e.g. money or a limited time budget. Thus, it is of interest to limit the amount of human intervention to as much as possible.

markus.goerlich-bucher@uni-a.de (M. Görlich-Bucher)

As an online learner, the *XCS Classifier System* (XCS) has gained notable attention in previously mentioned research initiatives. Originally designed as a reinforcement learning algorithm, there exist various derivatives for many machine learning settings, such as supervised classification tasks ([5]) or unsupervised learning tasks like clustering ([6]). A popular derivative is *XCSF*, a supervised function approximator ([7]). In an iterative manner, XCSF is confronted with an input X, calculates an approximation y and receives the corresponding ground truth $\widehat{y}$ for adapting its internal knowledge in order to provide better approximations in the future.

In this paper, we present an XCSF-based approach for learning to identify broken components in a intelligent system. More concrete, our solution focuses on limiting the amount of human supervision while maintaining an acceptable identification quality. In order to achieve this, we adapt the algorithmic internals of XCSF in order to reduce the number of *supervision* tasks according to [8]. Besides, in order to reduce the depency from accurately learning how internal component measurements might indicate a breakdown, we introduce a humber of so-called *triggers*. Triggers utilize several existing OC-metrics and other calculable measures. Furthermore, we provide a brief proof of concept to demonstrate our approach and discuss potential

The remainder of this paper is structured as follows. At first, we motivate our problem and introduce various relevant research works. Afterwards, our adapted XCSF as well as the Triggers implemented so far are presented. Our methodology is then evaluated using a real-world scenarios, before concluding with an outlook on possible future work.

2. Problem Statement

In the following, we motivate our problem statement using the architectural principles used in OC. However, the concepts presented in this paper can be applied to any other kind of intelligent system as well.

We assume an OC system S consisting of an *System under Observation and Control* (SuOC) as well as an *Multi Level Observer Controller*-Instance (MLOC). Intuitively, the SuOC represents some sort of machinery or complex technical system that is observed and controlled by the MLOC in order to work in a desired way. Typical examples for a SuOC include a smart factory floor, as used for evaluatory purposes later on, or a smart home environment. Each system is associated with some sort of utility measure U that can be used as a metric to assess the systems performance. Correspondingly, there may exist several variants of *disturbances* that affect U. A system is called robust if it manages to adapt to upcoming disturbances such that U remains over some acceptable performance threshold (or, at least, is able to return to an acceptable U shortly after a disturbance). A broader overview on this topic, as well as on the relations between disturbances and acceptable performance thresholds can be found in [9] .

S consists of various *components* $c \in |C|$. Hereby, a component acts as an abstraction for any kind of actuator or sensor that is used within the system context. As some sort of machinery or other physical device, each component situated in a real-world scenario will suffer from degradation or other *physical disturbances* from time to time, hence, disturbances that are not software-sided but situated within the system's components. We assume that a physical disturbance σ is not directly observable in terms of beeing noticed by S. However, we assume that an occurance of σ causes either permanent or at least temporary (if it is repaired) damage to the affected components in such ways that S is no longer able to perform on a level it was beforehand. Therefore, at least if no redundancy exists to compensate the broken component, σ affects S's utility U_{system}. Furthermore, depending on the type of a component, additional information such as sensor measurements or some sort of *local* utility measure U_c might exist. Again, these measurements could be affected by a breakdown and therefore might differ to those gained from an undisturbed component.

Finally, we assume that some sort of human expert - e.g. an repair worker or some sort of engineering staff - exists. The human expert has the corresponding expert knowledge to assess if a component is broken or not, hence, to deliver the ground truth to S, if necessary.

The overall problem can be summarized as follows: We intend to continuously assess the overall state of S in order to identify disturbances, or, in other words, to identify broken components. Hereby, we need to call upon a human expert in order to gain some ground truth on a component's state from time to time. As we want to reduce the overall costs, we seek to limit the amount of supervision as far as possible. Iteratively, S should be able to build a solid knowledge base for deciding if a component is broken, based on actual measurements from within the component or metrics generally available to S.

3. Related Work

A conceptual term closely related to this work is *self-healing*, altough it mostly focuses on software-sided disturbances (e.g. [10]). Few work exists on healing actual hardware-related failures (e.g. [11], [12]). However, self-healing differs from our work such that we solely focus on identifying disturbances, not healing (or repairing) them. Another relevant concept in OC (and other initiative for building intelligent systems as well) is *robustness*. Robustness describes the ability of a system to withstand or compensate disturbances while preserving a certain degree of functionality. An approach to quantify robustness was published in [9]. The concept of robustness is quite relevant to our work, as the identification of hardware failures might allow a system to proactively take measures to remain robust. On the other hand, we utilize the idea of a utility measure from [9] for the triggers in our own approach. Finally, the broader field of *Predictive Maintenance* (PdM) appears quite relevant as well. Plenty of PdM approaches make use of machine learning methods to either identify failures or predict future failure states. We refer to [13] for a more extensive introduction on this topic.

4. Methodology

As already mentioned in the introduction, our approach uses an adapted XCSF for learning under limited supervision, as firstly introduced in [8]. Intuitively, the XCSF is embedded in some sort of *control mechanism* (CM), e.g. the Multi-Layer Observer Controller (MLOC) architecture used in OC. Using several data sources from the components located in the observed and controlled system S (e.g. sensor data or previously mentioned triggers), as well as some external *supervisor* (e.g. a human expert), XCSF learns to identify broken components in an iterative manner. We suspect that the CM is able to assess S's overall performance in form of some *utility measure*. Depending on the actual scenario, we furthermore assume that the CM is be able to assess the performance of a single component. Besides, if S is able to assess the system utility and/or individual component utilities, we assume that there exist known utility boundaries until which S behaves in a desired state. We refer to [9] for a broader explanation of these boundaries. In the remainder of this section, we discuss the data sources that can be utilized and explain the XCSF implementation as well as the trigger implementations used throughout this work.

4.1. Representation and Data Sources

Depending on the application scenario, two types of data sources that can be used to learn from may exist: Actual measurements gathered from S's components as well as data gathered from the triggers S provides. Intuitively, it cannot be determined in advance how measurements may look like. For the scope of this work, we assume that the measurements gathered from the components are real-valued, however, it is also conceivable that more complex data sources exist (e.g. cameras return images). Trigger data, on the other hand, is represented binarily: Either a trigger is active or it is not.

4.2. Adapted XCSF

XCSF is an evolutionary online-learner for function approximation. On each discrete timestep, XCSF is confronted with a *situation description* (also called sigma in XCS-terminology) x, calculates an approximation y, and is presented with the actual correct prediction $\widehat{y}$. The XCSF implementation used throughout this work is based on the Algorithmic Description by [14] with XCSF-specific enhancements described in [7]. The *Min-Percentage representation*, originally introduced by Dam et al. [15], was used within the classifier conditions to represent real-valued data. Besides, the adaptions for limited supervision as described in [8] are included. The overall architecture as well as the learning process is described in the following.

XCSF consists of a *population* $[P]$, representing XCSF's knowledge base in form of *classifiers*. A classifier contains a *condition* representing the parts of the problem space the classifier covers, as well as various parameters. The original XCS described in [14] uses *tenary* conditions, that is binary conditions extended with a *do not care*-operator $\#$. Later implementations use different variants of real-valued conditions (cf. [16], [15]. As our problem space consists of both real-valued data (measurements) as well as binary data (triggers), the classifiers used in our implementation consist of a real-valued part, as well as a tenary part. Each classifier $cl_j \in [P]$ is associated with a linear approximation $h_j(x)$ estimating the *prediction* of the classifier for a given input x. The weights of $h_j(\cdot)$ are regarded as parameters of the classifier. Furthermore, a classifier contains a prediction error ϵ_j, estimating the error in h_j and a fitness value F_j. As the functionality of the linear approximation is of subordinate relevance for this work, we refer to [7] for a detailed description. Finally, each classifier is associated with a number of various *bookkeeping* parameters, for example its *experience*, that is, the number of match sets a classifier was part of.

Upon each discrete timestep t, XCSF is confronted with an input x. XCSF's $[P]$ is scanned for classifiers *matching* x. The hyperrectangular part of classifier matches iff for each $x_i \in x$, $l_i < x_i < u_i$ with l_i beeing the lower boundary and u_i beeing the upper boundary, respectively. The tenary part matches iff $x_i = c_i$ or $c_i = \#$ with $\#$ beeing the previously mentioned do not care-operator.

All classifiers matching x are added to the *match set* $[M]$. If no matching classifiers could be found, a *covering mechanism* is invoked. XCSF's original covering mechanism creates a matching condition by generating boundaries for each x_i in x with $l_i = x_i - R[0, s_0)$ and $u_i = x_i + R[0, s_o)$. Intuitively, $R[0, s_0)$ returns a random number between 0 and the *maximum spread* s_0. The classifiers parameters (e.g. the weights of $h_i(x)$, ϵ_j, F_j) are set to predefined values. For now, we do not replace the internal covering mechanism by an *external covering* as suggested in [8].

The covered classifier is added to both $[P]$ as well as $[M]$. Afterwards, $h_j(x)$ of each cl_j in $[M]$ is calculated. The results are averaged weighted with corresponding classifiers fitness values and returned as XCSF's prediction y. Afterwards, XCSF needs to decide if a supervision is necessary as explained in [8]: If the average experience of all classifiers lies below a certain predefined threshold θ_{exp} , the external supervision is called. The supervision provides the correct prediction $\widehat{y}$ which is used by XCSF to *update* the parameters of all classifiers in $[M]$. As the update cycle is not relevant for the understanding of this work, we again refer to [14] for a more detailed description.

On some timesteps, XCSF invokes a *genetic algorithm* (GA) after executing the update procedure. The GA selects two classifiers from $[M]$ and generates two offspring classifiers. With a certain probability, a crossover operator is applied to the boundaries of the offspring's condition. Also, again with a certain probability, a mutatation operator is used on the boundaries. Altogether with a *subsumption* procedure as well as a *deletion* scheme, the GA is responsible for some *evolutionary pressures* that allow XCSF to evolve populations of accurate classifiers ([17]). At this point, we refer to the corresponding literature ([14], [7]) for a more detailed explanation of XCSF's GA-related algorithmical structure.

As another approach for reducing the number of supervision calls, [8] suggest a *supervision cache* for XCSF. The idea of the supervision cache is to preserve knowledge for already supervised parts of the problem space. Due to XCSF's limitation of the population size, it is possible that previously learned classifiers are removed from $[P]$. The corresponding knowledge is lost, and when XCSF is confronted with an input situated in the corresponding part of the problem space, it needs to call on the supervision once again. Besides, it might happen that the GA produces offspring that demands supervision for parts of the problem spaces that was supervised before. The supervision cache acts as some sort of proxy for caching all performed supervisions in order to reduce supervision calls for already supervised situations. Each time the supervision is used, the cache saves both the situation as well as the correct prediction returned from the supervisor. The *manhattan distance* (md) is used to measure the similarity between an input x and previously cached supervisions:

$$md(x, x') = \sum_{i<n} |x_i - x'_i| \tag{1}$$

with x being the current input situation and x' the original situation for which the cache entry was created. The cache then returns the nearest matching supervision x' if $md(x, x') < \theta_{caching}$ with $\theta_{caching}$ being a predefined similarity threshold in order to avoid using situations that are too distant to the current input. It should be noted that the cache is only applied to the real-valued part of the problem space. This is due to the fact that the manhattan distance is not reasonably applicable to the binary part of the problem space and might skew the similarity calculated for the real-valued part.

4.3. Trigger Implementations

As mentioned earlier, the idea of the triggers is to reflect already existing metrics of the surrounding CM as well as to simplify potentially complex measurements. We identified and implemented four triggers of different complexity within the scope of this work, which will be explained in the following.

The *System-level utility trigger* T_S is based on the violation of the *acceptance space* boundary of the system S according to ([9]):

$$T_S(t) = \begin{cases} 1, & \text{if } U_t < \Theta_{acc} \\ 0, & \text{else} \end{cases} \tag{2}$$

Hereby, t is the current discrete timestep, Θ_{acc} the acceptance space boundary and U_t the utility of S at timestep t.

The *Component-level utility trigger* T_C is similar to T_S. The functionality follows the system level utility trigger so far. However, for this trigger, we assume that S is able to assess each component c's individual utility U_t^c as well as to provide an acceptance space boundary $\Theta_a cc_c$.

$$T_C(c,t) = \begin{cases} 1, & \text{if } U_t^c < \Theta_{acc_c} \\ 0, & \text{else} \end{cases} \tag{3}$$

The applicability of the component-level utility trigger might depend on the underlying SuOC: Not every scenario would feature component-wise utility measures.

The *external trigger* T_E formalizes the idea of a user filing some sort of maintenance/problem report after encountering errorous hardware in his environment: This could be, for example, a blue-collar worker utilizing some sort of machinery during his shift. Again, T_E will not necessarily be present in each conceivable scenario. Furthermore, we could suspect that the external trigger may be associated with a certain degree of uncertainty: The reporting user might just suspects a defect, while actually being not able to use the corresponding machinery in a correct way.

The *anomaly-based trigger* T_A is by far the most complex trigger. As mentioned before, we suspect that a component affected by a breakdown would also feature different measurements or metrics from its internal sensors. The idea of the anomaly-based trigger is to make use of a suspected measurable difference between data gathered during normal component function and data gathered during a breakdown. In order to do so, each time the supervision returns an $\widehat{y}$ that indicates a breakdown, the first n percent of the corresponding component's measurements are added to a so-called *normal data pool*. Afterwards, all measurements in the normal data pool are used to train a *One-Class Support Vector Machine* (SVM) ([18]). The idea of a One-Class SVM is to learn a decision boundary around already known data. Future data points which do not lie within these boundaries are considered to be *outliers* or anomalies. Therefore, T_A is activated when the trained SVM considers an input as an outlier. We use the `SGDOneClassSVM` Implementation from scikit-learn ([19]), as it allows to *warm-start* the SVM (therefore, to learn in an iterative, online manner).

5. Evaluation

5.1. Evaluation Scenario

The overall scenario is structured as follows. Our SuOC consists of 5 components in form of identical *production machines* producing some identical parts (e.g. for later processing in another area of the factory). The number of produced parts per time can be considered as the utility of the overall system. Accordingly, if a machine fails, the overall utility of the system drops below the acceptance threshold. Besides, we assume that the MLOC instance encapsulating the SuOC is able to assess the individual utility measures of each machine (therefore, the number of parts per time for each machine seperately). Furthermore, each machine delivers various internal measurements that can be used to assess the machines state. Additionally, no configuration changes (e.g. a changed system utility boundary due to a higher demand of parts) or other disturbances except actual machine breakdowns will take place, resulting in a quite simple evaluation scenario. We used the Azure AI Predictive Maintenance Dataset [1] for simulating the data measurements from the individual machines. The CSV-files in the dataset were preprocessed such that a single CSV-file exists for each machine (containing measurements from installation until breakdown in chronologial order) in the dataset. Besides, incomplete traces (that is, machines without breakdowns and machines that were repaired although no breakdown happened) were removed. Furthermore, the *error* column was removed, therefore, the only measurements available to XCSF are *volt*, *rotate*, *pressure* and *vibration*. This results in a total of 672 machines. The 5 components of the SuOC are equipped with 5 uniformly chosen machine CSV-files. If a component fails (that is: the CSV reaches its last row), it is replaced by another CSV. In an iterative manner, XCSF is confronted with the measurements for a machine as well as the corresponding calculated metrics. Furthermore, we assume that our supervision does not make mistakes, that is: If XCSF asks for supervision, the received ground truth is calculated based on the current system state. We evalute three different scenarios: 1) A naive XCSF without the limited supervision improvements, using only the Azure AI-measurements as a data basis (XCSF-M), 2) A limited XCSF with Caching using only the measurements (XCSF-C-M) and 3) A limited XCSF with Caching using the measurements as well as the trigger data (XCSF-C-MT). We conducted 30 repetitions with different fixed random seeds for each scenario. The parametrisation of the XCSF itself as well as the limited supervision mostly follows [8]:

[1] `https://www.kaggle.com/datasets/arnabbiswas1/microsoft-azure-predictive-maintenance`

$N = 800$, $\beta = 0.1$, $\alpha = 0.1$, $\nu = 5$, $\theta_{\text{GA}} = 48$, $\chi = 0.8$, $\mu = 0.04$, $\theta_{\text{del}} = 50$, $\delta = 0.1$, $\theta_{\text{sub}} = 200$, $\eta = 1.0$, $\epsilon_I = 0.0$, $F_I = 0.01$, $s_0 = 0.1$, $m = 0.2$, $\epsilon_0 = 1$ $P_{wildcard} = 0.33$, $s_{ext} = 0.25$, $\theta_{exp} = 3$, $\theta_{caching} = 0.2$

5.2. Results

The average system error as well as the supervision calls (for XCSF-C-M and XCSF-C-MT) are tested on significance. The shapiro-wilk-test was used in order to assess if we can suspect the data to be normal distributed. If so, a paired t-test was used for comparison, otherwise, a two-sided Wilcoxon rank-sum test was used. For all tests, an alpha-level of 0.05 was used. XCSF-M performed significantly better than both XCSF-C-M (Wilcoxon rank-sum test, p-value: 0.99) as well as XCSF-C-MT (Wilcoxon rank-sum test, p-value: 0.99). Also, XCSF-C-M performed significantly better dann XCSF-C-MT (Wilcoxon rank-sum test, p-value: 0.99). XCSF-C-M needed significantly more supervision calls (Wilcoxon rank-sum test, p-value: 0.99). The average system error as well as the needed supervision calls, both averaged over 30 runs are shown in Table 1. Figure 1 shows the average system error for all experiments over the course of the evaluation.

	System Error	Supervision Calls
XCSF-M	0.023 ± 0.002	-
XCSF-C-M	0.070 ± 0.018	247.40 ± 28.98
XCSF-C-MT	0.108 ± 0.023	1596.90 ± 103.19

Table 1
Results of the evaluation. System Error is equaled over the whole run, the calls are absolute values.

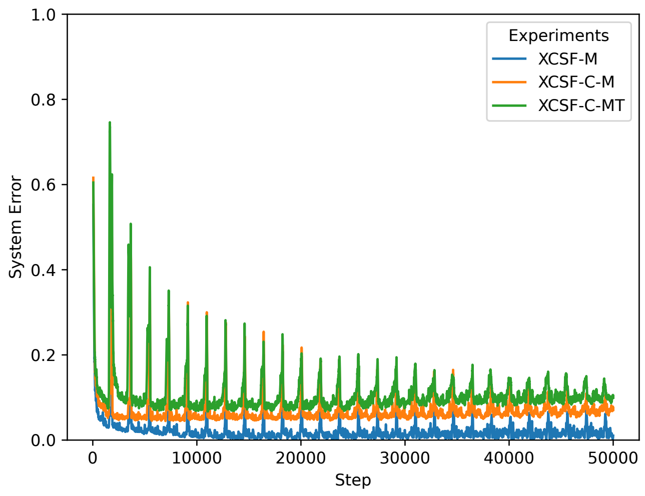

Figure 1: Averaged system error of all 30 repetitions for course of the evaluation, smoothed over 50 steps.

6. Discussion

Figure 1 shows an interesting behaviour: All XCSF-variants show periodic outliers in form of higher system errors. We suspect that the highly imbalanced data in combination with the order XCSF is confonted with the data might be responsible for this: After long periods of normally-labeled data, very short periods of failure-labeled data occur. The significance tests show that a naive XCSF is able to outperform the variants under limited supervision. However, this does not come to much as a surprise, as the latter have much less data to learn from due to the limitation. As [8] already stated, in scenarios where each supervision causes costs, a slightly worser performance can be an acceptable compromise for a reduced number of supervision calls.

Another interesting aspect is the worser performance of XCSF-C-MT compared to XCSF-C-M: Intuitively, one would suspect that an XCSF that has direct access to trigger derived from the actual system state would achieve better results than a variant only using measurement data. This is apparently not the case, also, the number of supervision calls of XCSF-C-MT by far exceeds the number of calls necessary for XCSF-C-M. A possible explanation could be that the more complex problem space of XCSF-C-MT (8 dimensions in contrast to 4 dimensions for XCSF-C-M) outweighs the benefits of the trigger data.

Up to this point, one might ask why XCSF is used as an algorithm here - as the setting is basically a binary classification task, a standard XCS for single-step problems would have been a valid choice as well. However, one major advantage of XCSF is that the prediction could be interpreted as a probability for an input belonging to the failure class (or to the normal data class). This aspect yields a quite interesting question for further research: One the one hand, we seek to limit the amount of supervision as much as possible. On the other hand, in more complex evaluation scenarios than this used during this paper, a larger amount of supervision might be necessary to adapt a classifier's prediction such that it adequately represents the failure probability.

7. Conclusion & Future Work

In this work, we proposed a novel approach on how systems from the broader domain of intelligent systems can identify hardware-related disturbances in their system context. We introduced several triggers derived from measurements and existing system metrics and integrated them in a XCSF-variant that is able to learn under limited supervision. We evaluated our approach using a simple proof of concept. The results indicate that further research is necessary in order to evaluate if the proposed methodology is applicable in more complex scenarios.

We currently see three interesting aspects for further research. First of all, a broader evaluation with different application scenarios appears useful. Based on the evaluation provided in this work, a quite obvious application scenario would be a more sophisticated smart factory environment, involving different kinds of production machinery. In such settings, the metrics used for calculating the triggers might differ, depending on the influence various machines might have on each other. Thereby, the corresponding XCSF-instance additionally needs to learn how the overall system context influences the trigger decisions in order to assess if a trigger can be used as a reliable source for properly identifying a component's physical state. Besides, any application scenario that involves intelligent systems consisting of sensors and actuators appears feasible. Exemplary scenarios involving more simple components than production machinery are sensor networks ([12]) as well as smart home/environment appliances ([20]). The actual behaviour of disturbances (e.g. sudden breakdown in contrast to slowly degrading components) can differ depending on the application scenario or the type of component. The triggers and measurements used right now only focus on a component's current state. However, it is conceivable to introduce a few more triggers that can be used to identify several disturbance characteristics in order to enlarge the potentially usable data presented to XCSF.

Another notable idea would be an investigation on how both the measurements as well as the triggers affect the performance of the individual learners. Depending on the scenario, there could be triggers that are not necessary for identifying failures. Intuitively, one would suspect that XCSF would evolve classifiers that feature a do not care-operator for the corresponding trigger. Finally, it must be investigated how various data-related aspects can influence the applicability of the whole approach. It is conceivable that more realistic application scenarios may feature more durable components compared to the evaluation provided in this work, resulting in an even more imbalanced dataset, which might affect XCSF's disturbance identification capabilities. A brief discussion on imbalanced data in XCS-research can be found in [21]. Also, an application scenario might feature some sort of concept drift or shift ([22]), e.g. through a slightly changed hardware configuration. Due to the limitation of supervision received by XCSF, it is conceivable that such a drift will affect the prediction quality without noticing, as (at least after building a stable classifier population) no ground truth is received anymore. A simple approach to compensate this aspect would be to introduce some sort of durability parameter for classifiers: From time to time, XCSF could call for supervision for classifiers with a specific age in order to validate if their prediction is still correct.

References

[1] C. Müller-Schloer, S. Tomforde, Organic Computing-Technical Systems for Survival in the Real World, Springer, 2017.

[2] J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (2003) 41–50. doi:10.1109/MC.2003.1160055.

[3] A. Stein, S. Tomforde, J. Botev, P. R. Lewis, Lifelike computing systems, in: Proceedings of the Lifelike Computing Systems Workshop (LIFELIKE), 2021.

[4] M. Görlich, A. Stein, J. Hähner, Towards physical disturbance robustness in organic computing systems using MOMDPs, in: 2019 Intelligent Systems Workshop in Workshop Proceedings of the 32nd International Conference on Architecture of Computing Systems, ARCS 2019, VDE, 2019, pp. 135–142.

[5] A. Orriols-Puig, E. Bernadó-Mansilla, A further look at UCS classifier system, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '06, 2006, pp. 8–12.

[6] L.-D. Shi, Y.-H. Shi, Y. Gao, L. Shang, Y.-B. Yang, XCSc: A novel approach to clustering with extended classifier system, International Journal of Neural Systems 21 (2011) 79–93.

[7] S. W. Wilson, Classifiers that approximate functions, Natural Computing 1 (2002) 211–234.

[8] M. Görlich-Bucher, J. Hähner, XCSF under limited supervision, in: Proceedings of the 2022 Genetic and Evolutionary Computation Conference Companion, GECCO '22, 2022.

[9] S. Tomforde, J. Kantert, C. Müller-Schloer, S. Bödelt, B. Sick, Comparing the effects of disturbances in self-adaptive systems-a generalised approach for the quantification of robustness, in: Transactions on Computational Collective Intelligence XXVIII, Springer, 2018, pp. 193–220.

[10] J. Schmitt, M. Roth, R. Kiefhaber, F. Kluge, T. Ungerer, Using an automated planner to control an organic middleware, in: 2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems, IEEE, 2011, pp. 71–78.

[11] E. Maehle, W. Brockmann, K.-E. Grosspietsch, A. El Sayed Auf, B. Jakimovski, S. Krannich, M. Litza, R. Maas, A. Al-Homsy, Application of the organic robot control architecture orca to the six-legged walking robot oscar, in: Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 517–530.

[12] M. Jänicke, B. Sick, P. Lukowicz, D. Bannach, Self-adapting multi-sensor systems: A concept for self-improvement and self-healing techniques, in: 2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops, IEEE, 2014, pp. 128–136.

[13] T. P. Carvalho, F. A. Soares, R. Vita, R. d. P. Francisco, J. P. Basto, S. G. Alcalá, A systematic literature review of machine learning methods applied to predictive maintenance, Computers & Industrial Engineering 137 (2019). URL: https://www.sciencedirect.com/science/article/pii/S0360835219304838. doi:https://doi.org/10.1016/j.cie.2019.106024.

[14] M. V. Butz, S. W. Wilson, An algorithmic description of XCS, Soft Computing 6 (2002) 144–153.

[15] H. H. Dam, H. A. Abbass, C. Lokan, Be real! XCS with continuous-valued inputs, in: Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation, GECCO '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 85–87. URL: https://doi.org/10.1145/1102256.1102274. doi:10.1145/1102256.1102274.

[16] S. W. Wilson, Get real! XCS with continuous-valued inputs, in: International Workshop on Learning Classifier Systems, Springer, 1999, pp. 209–219.

[17] M. Butz, T. Kovacs, P. Lanzi, S. Wilson, Toward a theory of generalization and learning in XCS, IEEE Transactions on Evolutionary Computation 8 (2004) 28–46. doi:10.1109/TEVC.2003.818194.

[18] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, R. C. Williamson, Estimating the support of a high-dimensional distribution, Neural computation 13 (2001) 1443–1471.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[20] F. Allerding, H. Schmeck, Organic smart home: Architecture for energy management in intelligent buildings, in: Proceedings of the 2011 Workshop on Organic Computing, OC '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 67–76. URL: https://doi.org/10.1145/1998642.1998654. doi:10.1145/1998642.1998654.

[21] M. Nakata, W. Browne, T. Hamagami, K. Takadama, Theoretical XCS parameter settings of learning accurate classifiers, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, 2017, pp. 473–480.

[22] S. Wang, S. Schlobach, M. Klein, What is concept drift and how to measure it?, in: International Conference on Knowledge Engineering and Knowledge Management, Springer, 2010, pp. 241–256.