XCS FOR SELF-AWARENESS IN AUTONOMOUS COMPUTING SYSTEMS

DISSERTATION

A thesis submitted to the
FACULTY FOR COMPUTER SCIENCE, ELECTRICAL ENGINEERING AND MATHEMATICS
of
PADERBORN UNIVERSITY
in partial fulfillment of the requirements
for the degree of *Dr. rer. nat.*

by

TIM HANSMEIER

Paderborn, Germany
Date of submission: June 2023

SUPERVISORS:
Prof. Dr. Marco Platzner

REVIEWERS:
Prof. Dr. Marco Platzner
Prof. Dr. David Andrews
Prof. Dr. Sybille Hellebrand

ORAL EXAMINATION COMMITTEE:
Prof. Dr. Marco Platzner
Prof. Dr. David Andrews
Prof. Dr. Sybille Hellebrand
Jun.-Prof. Dr. Sebastian Peitz
Dr. Heinrich Riebler

DATE OF SUBMISSION:
June 2023

Never confuse education with intelligence,
you can have a PhD and still be an idiot.

— Richard P. Feynman

ACKNOWLEDGMENTS

First and foremost, I want to express my gratitude to Marco for all the support, guidance, and opportunities he has provided to me – not only during my time as a PhD student but also during my (under-)graduate studies. It is unlikely that I would have started a PhD project with someone else. Further, I want to thank Prof. Dr. David Andrews and Prof. Dr. Sybille Hellebrand for their effort in reviewing this thesis, and Prof. Dr. Sebastian Peitz and Dr. Heinrich Riebler for taking the time to participate in the committee.

Doing research is not a one person's job. As such, I acknowledge the support of Mathis Brede, who has developed an XCS library in his Bachelor's thesis and extended it during his time as a student research assistant. His work contributed significantly to smooth experimental evaluations and allowed me to focus on more fundamental research aspects. Also, I want to thank all members of the computer engineering group that I have met over the years for providing a productive and pleasant working atmosphere. Specifically, I want to mention Christian Lienen and Linus Witschen, with whom I spent coffee and tea breaks that really deserved to be called *breaks*. They have been entirely unrelated to research and totally unproductive (the breaks, not Christian and Linus). Outside of our research group, I want to thank the entire food services department of the Studierendenwerk Paderborn for flawlessly feeding me lunch since 2014.

Finally, I want to thank my parents, who have (literally) supported me my entire life and enabled me to cope on my own, i.e., autonomously, with both academia and the real world. As argued by the thesis at hand, this is a skill I can be most grateful for.

ABSTRACT

The design paradigm of computational self-awareness tackles the increasing complexity in modern computing systems by moving design-time decisions to the runtime and into the system's responsibility. The required autonomous and adaptive behavior is often implemented by online learning techniques. Frequently proposed is the use of learning classifier systems, most notably their popular variant XCS. It combines reinforcement learning with a genetic algorithm to evolve an interpretable population of rules (termed classifiers). However, XCS has rarely been applied in real-world applications, and research is lacking on how XCS can be successfully applied to implement autonomy and adaptivity in practical application scenarios. This thesis makes a step toward bridging this gap in research. It presents the first experimental comparison of explore/exploit strategies for XCS, which allow XCS to autonomously decide if new knowledge about the environment needs to be gathered or if the existing knowledge can be exploited. An automated parameter optimization equips system designers planning to employ XCS with a set of useful hyperparameter configurations for each strategy. To fulfill safety guarantees, this thesis introduces the concept of forbidden classifiers. These special classifiers are hand-crafted, utilizing the interpretability of XCS's rule base, and prevent the selection of actions that violate safety requirements. The experimental evaluation shows that forbidden classifiers enable XCS to find a problem solution in a shorter time, with fewer classifiers, and with a lower computational burden when compared to XCS with an external safety shield. Finally, as an example of a practical application scenario, a case study investigates using XCS to control a CPU's frequency. While XCS consistently achieves better results than tabular Q-learning, it depicts a learning behavior that vastly differs from the behavior observed in the artificial problem environments commonly used in XCS research. As such, the case study highlights the need to develop mechanisms that enable XCS to automatically detect environmental characteristics that prevent it from deriving an optimal solution and take appropriate countermeasures, e.g., through hyperparameter self-configuration.

ZUSAMMENFASSUNG

Das Entwurfsparadigma der Computer-Selbstwahrnehmung (*Computational Self-Awareness*) begegnet der zunehmenden Komplexität moderner Computersysteme, indem es Entscheidungen zur Entwurfszeit in die Laufzeit und in die Verantwortung des Systems selbst verlagert. Das benötigte autonome und adaptive Verhalten wird häufig durch Online-Lerntechniken realisiert. Häufig wird der Einsatz von lernenden Klassifizierersystemen (*Learning Classifier Systems*) vorgeschlagen, insbesondere deren populärste Variante XCS. Es kombiniert Techniken des bestärkenden Lernens mit einem genetischen Algorithmus, um eine interpretierbare Population von Regeln (sogenannte *Classifiers*) zu entwickeln. Allerdings wurde XCS bisher nur selten in realen Anwendungen eingesetzt, und es fehlt an Forschung darüber, wie XCS erfolgreich verwendet werden kann, um Autonomie und Adaptivität in praktischen Anwendungsszenarien zu realisieren. Diese Dissertation macht einen Schritt hin zur Schließung dieser Forschungslücke. Sie präsentiert den ersten experimentellen Vergleich von *Explore/Exploit*-Strategien für XCS, welche es XCS erlauben, autonom zu entscheiden, ob neues Wissen über die Umgebung gesammelt werden muss oder ob das vorhandene Wissen genutzt werden kann. Eine automatisierte Parameteroptimierung gibt Systemdesignern, die den Einsatz von XCS planen, für jede Strategie eine Reihe von nützlichen Hyperparameterkonfigurationen an die Hand. Um Sicherheitsgarantien zu erfüllen, führt diese Dissertation das Konzept der verbotenen Klassifizierer (*Forbidden Classifiers*) ein. Diese speziellen Klassifizierer werden von Hand erstellt, wobei die Interpretierbarkeit der XCS-Regelbasis genutzt wird, und verhindern die Auswahl von Aktionen, die gegen Sicherheitsanforderungen verstoßen. Die experimentelle Auswertung zeigt, dass XCS mit verbotenen Klassifizierern eine Problemlösung in kürzerer Zeit, mit weniger Klassifizierern, und mit einem geringeren Rechenaufwand finden kann als XCS mit einem externen Sicherheitsschild. Schließlich wird in einer Fallstudie der Einsatz von XCS zur Steuerung der CPU-Frequenz untersucht. Während XCS durchgängig bessere Ergebnisse als tabellarisches Q-Learning erzielt, zeigt es ein Lernverhalten, das sich stark von dem Verhalten unterscheidet, das in den künstlichen Problemumgebungen beobachtet wird, welche üblicherweise in der XCS-Forschung verwendet werden. Die Fallstudie unterstreicht die Notwendigkeit, Mechanismen für XCS zu entwickeln, mit denen es automatisch solche Umgebungsmerkmale erkennen kann, welche die Ableitung einer optimalen Lösung verhindern. XCS kann dann entsprechende Gegenmaßnahmen ergreifen, z.B. durch Selbstkonfiguration der Hyperparameter.

AUTHOR'S PUBLICATIONS

[1] Mathis Brede, Tim Hansmeier, and Marco Platzner. "XCS on Embedded Systems: An Analysis of Execution Profiles and Accelerated Classifier Deletion." In: *Proceedings of the 2022 Genetic and Evolutionary Computation Conference Companion (GECCO'22)*. ACM, 2022, pp. 2071–2079. DOI: 10.1145/3520304.3533977.

[2] Tim Hansmeier. "Self-Aware Operation of Heterogeneous Compute Nodes Using the Learning Classifier System XCS." In: *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'21)*. ACM, 2021, pp. 1–2. DOI: 10.1145/3468044.3468055.

[3] Tim Hansmeier, Mathis Brede, and Marco Platzner. "Safe Learning with XCS via the Injection of Forbidden Classifiers." In: *SN Computer Science* (tbd). Invited submission. Currently under review.

[4] Tim Hansmeier, Paul Kaufmann, and Marco Platzner. "An Adaption Mechanism for the Error Threshold of XCSF." In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO'20)*. ACM, 2020, pp. 1756–1764. DOI: 10.1145/3377929.3398106.

[5] Tim Hansmeier, Paul Kaufmann, and Marco Platzner. "Enabling XCSF to Cope with Dynamic Environments via an Adaptive Error Threshold." In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO'20)*. ACM, 2020, pp. 125–126. DOI: 10.1145/3377929.3389968.

[6] Tim Hansmeier and Marco Platzner. "An experimental comparison of explore/exploit strategies for the learning classifier system XCS." In: *Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion (GECCO'21)*. ACM, 2021, pp. 1639–1647. DOI: 10.1145/3449726.3463159.

[7] Tim Hansmeier and Marco Platzner. "Integrating Safety Guarantees into the Learning Classifier System XCS." In: *Applications of Evolutionary Computation (EvoApplications 2022)*. **Nominated for Best Paper Award**. Springer, 2022, pp. 386–401. DOI: 10.1007/978-3-031-02462-7_25.

[8] Tim Hansmeier and Marco Platzner. "Autonomous Explore/Exploit Strategies for XCS: An Experimental Comparison." In: *Soft Computing* (tbd). Currently under review.

[9] Tim Hansmeier, Marco Platzner, and David Andrews. "An FPGA/HMC-Based Accelerator for Resolution Proof Checking." In: *International Symposium on Applied Reconfigurable Computing (ARC 2018)*. **Received Best Paper Award**. Springer, 2018, pp. 153–165. DOI: 10.1007/978-3-319-78890-6_13.

[10] Tim Hansmeier, Marco Platzner, Md Jubaer Hossain Pantho, and David Andrews. "An Accelerator for Resolution Proof Checking based on FPGA and Hybrid Memory Cube Technology." In: *Journal of Signal Processing Systems* 91.11 (2019), pp. 1259–1272. DOI: 10.1007/s11265-018-1435-y.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AC	Autonomic Computing
ACS	Anticipatory Learning Classifier System
CPS	Cyber-Physical System
DVFS	Dynamic Voltage and Frequency Scaling
E/E	Explore/Exploit
EC	Evolutionary Computation
EPICS	Engineering Proprioception in Computing Systems
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
IPF	Information Processing Factory
IPS	Instructions per Second
LCS	Learning Classifier System
LCT	Learning Classifier Table
MAPE-K	Monitor-Analyze-Plan-Execute loop with Knowledge
MIPS	Million Instructions per Second
OC	Organic Computing
ODA	Observe-Decide-Act
QoS	Quality of Service
RBML	Rule-based Machine Learning
RL	Reinforcement Learning
SEEC	SElf-awarE Computing
SuOC	System under Observation and Control

1

INTRODUCTION

Modern computing systems continue to become ubiquitous, with application domains ranging from large-scale data centers to tiny embedded computing systems. The different types of computing systems must often communicate with each other, e.g., in edge-fog-cloud networks [2], leading to complex interactions and a high degree of heterogeneity. Managing this complexity becomes increasingly challenging, especially with varying requirements and optimization goals, such as a variable availability and cost of electrical energy. In addition, computing systems employed in Cyber-Physical Systems (CPSs) deal with physical environments that can, at most, be partially specified at design-time and even change during operation. Hence, computing systems must be enabled to cope autonomously with uncertainties and adapt to environmental changes. This goal is inherent to several design paradigms [43, 74, 79], among them the concept of computational self-awareness [66]. The notion of self-awareness draws inspiration from psychology, as humans are considered to have the highest and most versatile level of autonomy and adaptivity. This is achieved by continuously reasoning about subjective experiences made in the past and translating them into decision strategies that are used in the future. The same approach is followed in the concept of computational self-awareness, where static design-time decisions are moved to the runtime and into the system's responsibility. This allows technical systems to cope autonomously with increasing complexities, unforeseen situations, and environmental changes.

Learning Classifier Systems (LCSs) [106] are frequently proposed for use in the design paradigm of computational self-awareness and related approaches [73, 104]. With the combination of Reinforcement Learning (RL) [99] and Evolutionary Computation (EC) [7], LCSs follow two lifelike learning mechanisms that promise high adaptivity. The most common LCS is XCS[1] [110], an online Rule-based Machine Learning (RBML) technique that relies on a dynamic population of rules, which are called classifiers. Each rule in the population proposes an action for the situations it matches. The usefulness of the rules is determined via RL, i.e., through interactions with the operational environment, while new rules are generated by a Genetic Algorithm (GA) that aims at generalizing the

1 Some researchers have backronymed XCS to stand for eXtended Classifier System. This thesis, however, sticks to the original meaning and does not treat XCS as an acronym.

rules such that they apply in as many situations as possible. Through the combination of reinforcement learning with a genetic algorithm, XCS is able to evolve a rule population of a minimal size that can determine the best-suited action for each encountered situation. Furthermore, since XCS is an online learning algorithm, it can adjust to the specifics of a deployment environment and environmental changes during operation. Therefore, XCS can be applied to implement autonomous and adaptive behavior in technical systems. In addition, its evolved rule base depicts a high degree of interpretability, allowing human operators to analyze and validate the behavior that an autonomous system has learned.

However, even though XCS seems to be a natural fit to implement autonomy and adaptivity in self-aware systems, it is rarely applied for such use cases. Instead, XCS is commonly developed and evaluated in artificial toy problem environments only, while descendants of XCS are primarily applied for supervised classification tasks [41, 80]. The rarity of works that employ XCS in autonomous systems to tackle practical application scenarios is related to a lack of research on how XCS can be applied in autonomous and adaptive real-world systems. This thesis aims at bridging the research gap by improving the amenability of XCS for use in autonomous systems and investigating the behavior of XCS in a real-world problem environment.

1.1 THESIS CONTRIBUTIONS

This thesis makes three key contributions that improve XCS's ability to be successfully deployed in self-aware computing systems that tackle practical application scenarios. It does so by investigating and improving the learning behavior of XCS and revealing characteristics of real-world application environments that must be considered before deployment.

Explore/Exploit Dilemma (Chapter 4). As every reinforcement learner, XCS faces the Explore/Exploit (E/E) dilemma [111] when deciding which action should be executed. On the one hand, XCS learns by trying various actions in the encountered situations to observe which is the best-suited action for a given situation (*exploration*). On the other hand, it should maximize the operational performance by always selecting the best action (*exploitation*). If the learner focuses too much on exploitation, it cannot reliably determine the best action, and the operational performance stays below its maximum. On the other hand, if the focus is primarily on exploration, non-optimal actions are deliberately chosen, and operational performance is also negatively impacted. Hence, the E/E decision must be taken based on the current learning progress. For XCS, very few E/E strategies that autonomously decide which way to pursue have been proposed in the existing research literature, even though such strategies are a requirement for employing XCS in autonomous and adaptive systems. In addition, the proposed E/E strategies have been evaluated in different problem environments and have never been compared to each other.

This thesis presents the first experimental comparison of four E/E strategies proposed in the research literature. The comparison takes place on several problem environments and also considers environmental dynamics, which make the E/E dilemma complex to solve. In addition, we employ an offline automated hyperparameter optimization to determine suitable hyperparameter configurations for each strategy. As such, the experimental study provides directions for future work on autonomous E/E strategies for XCS and equips XCS practitioners with guidelines on how to employ E/E strategies successfully.

Safety Guarantees (Chapter 5). Especially when used in CPSs, XCS must fulfill safety requirements during operation. However, due to its RL paradigm, it is prone to select safety-violating actions, most apparently during exploration. Traditional approaches for preventing safety violations add an external shield to the learner that constantly monitors the environmental states and actions. If a violation of safety requirements is detected, the shield steps in and sanitizes the action that would impact safety.

To directly equip XCS's rule base with safety guarantees, we developed the concept of *forbidden classifiers*. Such classifiers prevent the selection of specific actions in safety-critical situations. They are hand-crafted based on safety requirements and injected into the classifier population before deployment. To the best of our knowledge, this work is among the first to leverage XCS's rule-interpretability by systematically injecting domain knowledge. Our experimental evaluation shows that forbidden classifiers can implement safety guarantees just as an external shield but enable XCS to solve the evaluated problem environments more quickly, with fewer classifiers, and a lower computational burden. In contrast to XCS with an external shield, forbidden classifiers directly internalize the safety-critical knowledge and spare XCS from learning the same knowledge through interactions with the shield. This is also why forbidden classifiers can be used to inject domain knowledge and accelerate XCS's learning progress even when no safety requirements must be met.

Learning Behavior in Real-world Environments (Chapter 6). The majority of research works develop and evaluate XCS only in artificial toy problem environments. These well-controllable environments depict properties that allow XCS to evolve a minimally sized classifier population with accurate payoff predictions, making such environments ideal for studying and extending the learning mechanism of XCS. However, real-world environments often do not depict such ideal conditions, and so far, XCS has rarely been applied in practical application scenarios. One of the few domains where LCSs have turned out to be successful is the Dynamic Voltage and Frequency Scaling (DVFS) in CPUs [26, 98, 117].

In a case study, we also apply XCS for DVFS to study its behavior in a real-world problem environment that is known to be amenable to LCSs. We confirm the related works' observation that XCS uses its dynamic classifier generalization to outperform tabular Q-learning [98]. In addition, we provide the first extensive study of XCS's learning behavior in

this application scenario and find several differences against the behavior observed in common toy problem environments, such as an uneven allocation of classifiers to environmental niches. The characteristics of the evaluated application domain prevent XCS from evolving properly generalized classifiers, thereby limiting its capability to solve more complex problems. A manual hyperparameter tailoring based on theoretical insights on XCS leads to only minor improvements, highlighting the need to develop robust hyperparameter self-configuration mechanisms in future work.

1.2 THESIS ORGANIZATION

The remainder of this thesis is structured into six chapters and continues as follows:

- Chapter 2 presents the design paradigm of computational self-awareness, along with other concepts that aim at making technical systems more autonomous and adaptive. To fill the approaches with life, selected research works that equip systems with self-awareness capabilities are shortly presented.
- Chapter 3 provides extensive background on XCS, enabling readers unfamiliar with XCS to follow the thesis. In addition to an algorithmic description of XCS, interested readers can find a discussion of its internal interactions and working mechanisms. Finally, it is motivated why XCS is a promising candidate for deployment in self-aware systems.
- Chapter 4 experimentally compares four E/E strategies for XCS. The strategies are evaluated in static and dynamic problem environments, and an automated parameter optimization is conducted for each environment.
- Chapter 5 introduces the concept of forbidden classifiers. The use of forbidden classifiers is not only evaluated in the context of safety guarantees but also in the context of manual knowledge injection for bootstrapping the classifier population.
- Chapter 6 presents the case study in which XCS performs DVFS. In addition to the frequency control behavior, the learning behavior of XCS is extensively investigated.
- Chapter 7 concludes the thesis and outlines future work.

2

COMPUTATIONAL SELF-AWARENESS

Designing and operating modern computing systems is becoming an increasingly challenging task. For instance, the heterogeneity and parallelism found in modern state-of-the-art compute nodes might increase the potential to reach the best cost-per-performance ratio but also require more complex operation strategies. In addition, environmental dynamics, such as changes in the network topology of large distributed systems, must be accounted for while application requirements also become more diverse, with non-functional execution properties, most prominently energy consumption, gaining importance. Common static approaches typically cannot tackle such heterogeneous, complex, and uncertain environments. As one possible outcome, the design paradigm of self-aware computing has been proposed [66]. Computational self-awareness describes the property of a computing system to proactively gather information and knowledge about its internal state and environment to make adequate decisions. As such, self-aware computing systems can cope autonomously with dynamic or uncertain environments.

The concept of computational self-awareness draws inspiration from psychology, as humans are commonly considered to depict the most versatile level of autonomy and adaptivity. However, transferring the psychological notion of self-awareness into the domain of computing systems is not straightforward, which is why the term self-awareness is used ambiguously in computer science. This thesis sticks to the definition of computational self-awareness as developed in the Engineering Proprioception in Computing Systems (EPiCS) project [66]. It refines the core concepts from psychology, builds a design principle accessible to engineers and system designers, and aids a structured engineering process for adaptive systems. In addition, the architecture for self-aware computing systems can be used to systematically analyze and compare existing systems' self-awareness capabilities.

The chapter continues by presenting the core concepts of computational self-awareness in Section 2.1. Afterward, a reference architecture for self-aware computing systems is described in Section 2.2. Next, other concepts that are closely related to the notion of computational self-awareness are discussed in Section 2.3. Lastly, Section 2.4 fills the concepts with life and gives an overview of selected applications where aspects of computational self-awareness have been applied.

2.1 KEY CONCEPTS

The notion of computational self-awareness developed in the EPICS project is inspired by psychology and includes three key concepts that are frequently observed when discussing self-awareness in humans [65]:

- Self-awareness can be directed toward the inside of the individual or to the external environment. This results in the distinction of private and public self-awareness as outlined in Subsection 2.1.1.
- Self-awareness is a spectrum with a versatile range of capabilities. Subsection 2.1.2 describes five different layers of computational self-awareness inspired by Neisser [76], a psychologist who investigated the different self-awareness capabilities present in living beings.
- Self-awareness is a property not restricted to individual systems but can also be exhibited by collective systems. As described by Subsection 2.1.3, self-awareness can even emerge through the interaction of non-self-aware subsystems.

All aspects of self-awareness considered in this section only concern knowledge generation. Appropriate decision-making must make use of the obtained knowledge to take reasoned actions. In the framework developed in the EPICS project, self-expression is the behavior resulting from the knowledge acquired through self-awareness. The separation between self-awareness and self-expression, i.e., knowledge generation and decision-making, can aid the systematic analysis of self-aware computing systems, even though both aspects are often treated together. Self-expression is discussed in more detail in Section 2.2 in the context of a reference architecture for computational self-awareness.

2.1.1 *Private and Public Self-Awareness*

Psychology often distinguishes two forms of self-awareness, where one is directed primarily to the inside of the individual, while the other is directed to its surroundings. Lewis et al. [65] apply this distinction to the engineering domain and define the concept of private and public self-awareness. Private self-awareness is the ability to observe phenomena that are internal to the system, which means that they are typically externally unobservable and can only be perceived by the system itself, e.g., the battery level of a mobile system. The knowledge derived from these observations is thus private to the system and cannot be accessed from the outside except the system explicitly communicates it. Sensing internal phenomena requires internal sensors, an explicit building block of the reference architecture described in Section 2.2.

Public self-awareness is concerned with generating knowledge from phenomena external to the system. It is therefore based on the (subjective) perception of the environment by the system. It can include simple

aspects, such as the current physical state of the environment, but also more complex derived knowledge, such as the relationship to other systems or models of the own interaction with the environment. As such, the knowledge created through public self-awareness results from the experiences with the system's environment.

The distinction between private and public self-awareness highlights the need for systems to consider not only knowledge of their internal state but also of the current state of the environment and the interaction with it [64]. In the end, a self-aware system should dynamically and autonomously adapt to unforeseen circumstances, which is only possible if it considers both the external environment and the internal state. Furthermore, separating public and private self-awareness should allow system designers to reason about the cost and benefits associated with different types of knowledge generation in a more structured way [65].

2.1.2 *Levels of Computational Self-Awareness*

The capability of self-awareness is typically not considered an all-or-nothing phenomenon but a spectrum with different degrees of self-awareness. To capture this systematically, Neisser [76] has described five levels of self-awareness that can be observed in living beings. However, these levels do not straightforwardly transfer from psychology to computing, as they include notions of thoughts or feelings, which are undefined for technical systems. In addition, the levels build on each other, which is overly restrictive for the engineering and analysis of computing systems. In living beings, the different levels of self-awareness have emerged through evolution and are in line with the degree of "sophistication" associated with the being. In technical systems, however, different self-awareness capabilities can exist independently of each other, just as it seems fit by the system designers. To this end, Lewis et al. [65] have taken Neisser's levels of self-awareness and adapted them to be more suited to the domain of computing systems. These different levels of *computational* self-awareness are outlined below.

Stimulus-awareness. A stimulus-aware computing system can generate and act on knowledge based on current stimuli. However, the system has no knowledge of past or future stimuli and thus is restricted to reactive behavior. Stimulus-awareness is the most basic form of self-awareness. It is the only capability present in all self-aware computing systems, as sensing current stimuli is a necessary requirement for all other layers. Since stimuli can come from the inside and outside of a system, stimulus-awareness can be a form of private and public self-awareness.

Interaction-awareness. With interaction-awareness, a system is aware that other systems operate in the same environment and that its own actions and those of the other systems constitute interactions – both with each other and with the environment. Without interaction-awareness, a computing system can still perceive stimuli from other systems via its stimulus-awareness, but cannot distinguish it from the other environmen-

tal stimuli received. Consequently, the presence of other systems cannot be considered in the own decision-making. With interaction-awareness, stimulus received from other systems can be incorporated explicitly into the own knowledge base. A simple form of interaction-awareness could be that a system is considering the actions taken by other systems in its decision-making process, while more advanced forms can lead to explicit cooperation. Typically, interaction-awareness is considered a part of public self-awareness, as the interacting systems are situated in the shared external environment. However, it is also thinkable that a system interacts with itself, in which case it forms a type of private self-awareness.

Time-awareness. Adding a notion of time to a system makes it time-aware and enables it to reason about both the past and the future. With time-awareness, a system can have explicit knowledge of historical events and can use this knowledge to make predictions of phenomena that will likely occur in the future. With such anticipation, a self-aware system can proactively adapt or employ complex lookahead planning. This applies to the internals of a system and its external environment, making time-awareness a part of private and public self-awareness.

Goal-awareness. If a system can obtain knowledge about current goals, desired states, or constraints, it is said to be goal-aware. However, it must be noted that goal-awareness requires that the system is explicitly aware of its goals and can reason about its goals or even adapt them. Goals implicitly designed into the system do not fulfill the requirements for goal-awareness. Since every system is designed with a purpose in mind, every technical system could be considered goal-aware otherwise. When coupled with time-awareness, goal-awareness allows reasoning about likely future goals and permits considering them in the current decision-making. With interaction-awareness, a goal-aware system can reason about its own goals in relation to those of the other systems and unveil common or opposing goals. As such, goal-awareness can be both private and public self-awareness.

Meta-self-awareness. With meta-self-awareness, a system is aware of its own self-awareness and how the different levels are currently instantiated. The system can then reason about the cost and benefits associated with its current self-awareness capabilities. For instance, it can adapt how a layer is instantiated by switching to a different algorithm or deciding to (temporarily) disable a self-awareness layer completely. Meta-self-awareness is concerned only with the self-awareness capabilities of the system, which are inherently internal. Hence, meta-self-awareness is the only layer that can only be a form of private self-awareness.

All described layers of self-awareness can be instantiated independently, except for stimulus-awareness, which forms the fundamental layer of self-awareness. Despite the separation into different layers, the overarching criterion for computational self-awareness remains that the system can "obtain knowledge on an ongoing bases" [65]. Hence, a system with a fixed, implicit goal or hard-coded interactions with other systems is typically not considered goal- or interaction-aware, respectively.

2.1.3 *Collective Self-awareness*

The third key aspect of computational self-awareness acknowledges that self-awareness is not restricted to a singular system but can also be emanated by a collective of systems – even if it does not have a central point at which knowledge is located. In such a case, information about the global state is distributed, but the collective acts like it has a sense of its own state. However, a distributed self-aware system can still have a central point of control. With a centralized knowledge base, the interactions within a collective can likely be orchestrated more efficiently, while distributed knowledge that is stored at the points in the collective where it is needed likely results in a higher degree of robustness. However, the decision between a centralized and decentralized approach can be considered an architectural one and conceptually does not prevent any of the self-awareness capabilities from occurring [65].

In fact, a collective's subsystems do not need to be self-aware by themselves. In such a case, the self-awareness properties of the collective emerge solely through the interactions of its subsystems. It can also be the case that the subsystems are self-aware but that the collective shows a higher degree of self-awareness than the individual subsystems. As such, the concept of computational self-awareness can be applied both to the individuals of a collective and to the collective itself, i.e., the boundaries of the "self" can be arbitrarily chosen as it seems best for analysis or system design.

2.2 REFERENCE ARCHITECTURE

Based on the outlined notion of computational self-awareness, Chandra et al. [22, 64] have devised a reference architecture for self-aware systems. The architecture provides a common ground for system designers to engineer and compare systems with different self-awareness capabilities. When comparing or analyzing existing systems, the reference architecture can be used as a template for identifying common patterns. In the engineering process, the architecture facilitates a structured design process based on the psychological foundations of self-awareness. The reference architecture is based on the three key concepts of self-awareness and separates the processes associated with knowledge generation (self-awareness) from decision-making (self-expression). This distinction allows for a fine-grained analysis of systems and encourages engineers to try different implementations for the different components.

The reference architecture is shown in Figure 2.1 and comprises internal and external sensors and actuators, a (meta-)self-awareness engine, and a self-expression engine.

Sensors. To reason about itself and its environment, a system requires continuous observations of its internal state and the environment. Through its sensors, a self-aware system perceives both internal and external phenomena. In line with the distinction of private and public

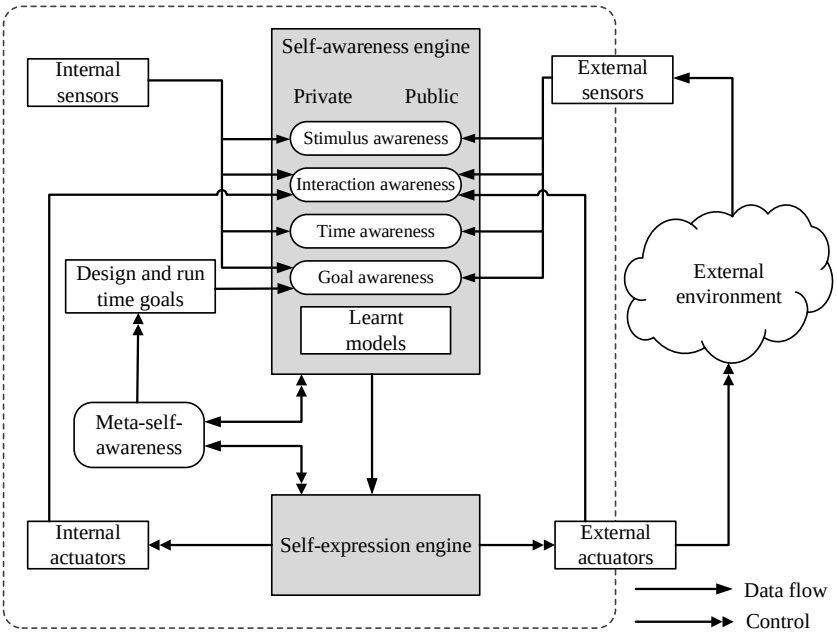


Figure 2.1: Reference architecture for self-aware computing systems [22, 64].

self-awareness, the internal sensors gather information related to the system's internal state and supply the self-awareness engine's private side. The external sensors collect data about the external environment and supply the public side of the self-awareness engine. The combination of both information flows allows a self-aware system to build models of itself and the environment according to its levels of self-awareness.

Actuators. The internal and external actuators are how a system can actively influence its environment or internal state, i.e., they form the "knobs" that the system can turn. Without actuators, a self-aware system cannot leverage its knowledge to achieve anything useful. With external actuators, actions that impact the external environment can be taken, including interactions with other systems. The internal actuators change the internal functionality of the system and can thereby pose interactions with itself. Reasoning about interactions via interaction-awareness requires considering own actions that are taken. The explicit data flow between the actuators and the interaction-awareness layer denotes this. However, the actions' results must be observed via the system's sensors.

Self-awareness. The core of the reference architecture is the self-awareness engine that encapsulates the processes which realize the different layers of computational self-awareness. Naturally, all layers of self-awareness rely on the sensory observation of the system. The reference architecture emphasizes that computational self-awareness is a process of knowledge generation and that static design-time knowledge is insufficient to qualify as self-aware. Hence, the current state of knowledge, i.e., the learned models, is separated from the processes of knowledge generation, i.e., the layers of self-awareness.

The goal-awareness layer uses design- and run-time goals to incorporate them into the knowledge base explicitly. Design-time goals can be high-level and directly related to the purpose of the system, e.g., completing a search-and-rescue mission in the case of a self-aware drone. The run-time goals, however, could change during the operation depending on the current internal state and the environment. For example, in the case of the self-aware drone, this could mean identifying regions of high interest, stabilizing flight if the weather is rough, or saving energy if the battery level is low. The meta-self-awareness engine can aid the system in identifying the costs and benefits of the different run-time goals and reasoning about the tradeoff they impose. Also, the meta-self-awareness engine can control the self-awareness capabilities of the system in consideration of the current run-time goals, e.g., by algorithm selection.

Self-expression. The self-expression engine uses the knowledge generated by the self-awareness engine to decide which actions to take. Since the self-expression engine controls internal and external actuators, actions can both modify the internal state or aim at impacting the external environment. The decision-making can range from simple static rules to complex decision processes dynamically generated with online learning. Naturally, the self-expression and self-awareness engines are tightly coupled, as the self-expression engine can only utilize the knowledge made available by the self-awareness engine. Furthermore, vice versa, the knowledge generation process of the self-awareness engine is affected by the exercised actions, as they influence the observed internal and external states. However, it can make sense to clearly distinguish between knowledge generation (self-awareness) and decision-making (self-expression) processes since it encourages system designers to evaluate different algorithms for self-awareness and self-expression independently of each other.

The reference architecture does not impose any restrictions on the way self-awareness and self-expression capabilities are implemented. Many different computational processes can implement such capabilities, but in many cases, online learning will be applied [109]. Also, not all reference architecture components must be present in a system to qualify as self-aware. As already mentioned, the levels of self-awareness can be used independently of each other, except for stimulus-awareness, which is always required. An overview of typical combinations of layers, termed patterns, is given by Chen et al. [23]. A system could also be equipped with internal sensors and actuators only. In such a case, the self-awareness and self-expression capabilities of the system are directed only toward its inside. This flexibility allows the boundaries of the reference architecture to be set as desired for the engineering or analysis task. In most cases, the boundaries of the architecture will encompass a single computing system with self-awareness and self-expression capabilities. However, the architecture can also encompass a collective, where the individuals of the collective have different sensors, actuators, or self-awareness and self-expression capabilities than the collective as a whole.

The main aim of the reference architecture is to encourage engineers to design self-aware systems in a structured way. For instance, the architecture can be used to describe and extend the degree of self-awareness already present in an existing system. All internal and external sensors and actuators would be collected, along with an analysis of the self-awareness and self-expression capabilities. This might reveal opportunities to add self-awareness or self-expression functionalities with minimal effort, e.g., because existing sensors or actuators can be re-used or useful knowledge is already being collected in the system. Also, the reference architecture provides a structured overview of what is missing and must be added to the system, e.g., virtual or physical sensors, or computational processes for knowledge generation.

2.3 RELATED CONCEPTS

This section presents four related concepts with objectives and design principles similar to the computational self-awareness paradigm developed in the EPICS project. Not all explicitly include a notion of self-awareness, but self-* properties such as self-adaptivity or self-configuration still play an important role. In all cases, the common goal is to tackle the complexity of technical systems by moving design-time decisions to the runtime and into the systems' responsibility. Presented are the concepts of Autonomic Computing (AC), Organic Computing (OC), and the SElf-awarE Computing (SEEC) framework, of which all three represent established approaches. As a younger design paradigm that is still under active development, the Information Processing Factory (IPF) is considered.

2.3.1 *Autonomic Computing*

Sparked by IBM in 2001, the Autonomic Computing (AC) initiative tackles the "software complexity crisis" [49]. Due to the large number of computing systems, which is increasing at an accelerated rate, increasing levels of heterogeneity and interconnectivity, and more demanding applications, the management of IT systems becomes more challenging. The software for managing the computing systems thus needs to be more complex, requiring more and better-skilled staff for operation. One of the main driving forces for IBM's AC initiative was the fear that the rise of computing systems would come to a halt simply because there are not enough IT operators available [25]. As an outcome, AC is proposed to enable self-management in computing systems.

Even though AC has a clear focus on the management of IT systems, it still draws inspiration from nature [49]. The autonomic nervous system of humans and other animals controls the heart rate, respiratory frequency, and body temperature without explicit control from the higher cognitive functions of the individual. The conscious brain is thereby relieved from

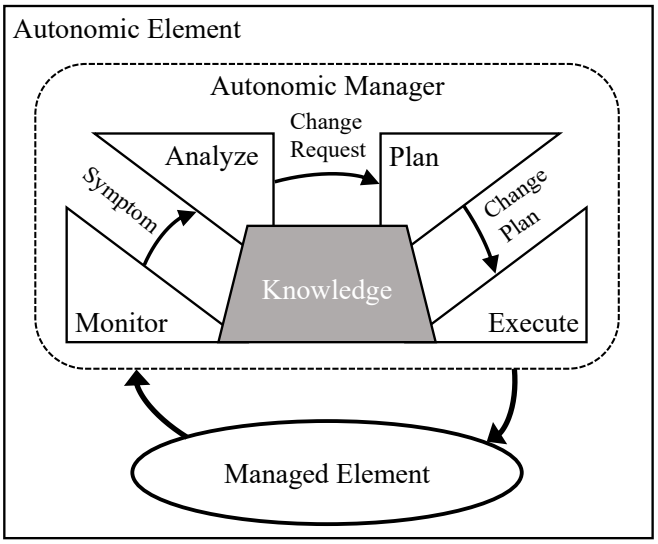


Figure 2.2: Structure of an autonomic element [5, 49]. The managed element is controlled by an autonomic manager with a MAPE-K loop.

such low-level control tasks and can focus on tasks more suited to its capabilities, such as complex reasoning on how to reach long-term goals. To enable self-management, the concept of AC includes four key capabilities, which traditionally are, or have been, the responsibilities of human operators.

- Self-configuration: The system determines its own configuration during runtime according to high-level policies. The policies define what behavior is desired but do not specify with which configurations this can be achieved.
- Self-optimization: The system continuously tunes its operating parameters to perform optimally.
- Self-healing: The system autonomously detects problems, e.g., software bugs or hardware failures, localizes them, and takes countermeasures for repair.
- Self-protection: The system can protect against malicious attacks or failures that the self-healing mechanisms cannot repair. Self-protection also includes anticipatory actions to protect against likely future incidents.

Even though not explicitly part of the four key capabilities, self-awareness is considered a requirement for self-management [79]. However, self-awareness is defined only as awareness of the own state and behavior, thereby representing the notion of private self-awareness developed in the EPICS project. The environmental awareness of a system is termed context awareness, which loosely corresponds to the notion of public self-awareness.

The main architectural primitive is an autonomic element, as shown in Figure 2.2. An autonomic element consists of one or more managed elements, equivalent to the non-autonomic elements found in traditional IT systems, and a single autonomic manager. For control, the autonomic manager applies a loop consisting of a monitor, analyze, plan, and execute phase (MAPE) [5]. The monitor function collects information on the status of the managed element, e.g., its configuration, throughput, or available capacity. The information is then aggregated, filtered, and searched for *symptoms* that need to be reported because they potentially require management actions. As such, the monitor phase encompasses more functionalities than just plain data measurements. The analyze function takes the information the monitor function provides and determines if changes need to be made. In this stage, complex information processing, such as forecasting, can be applied. The plan function takes the change request from the analyze phase and generates a change plan that encompasses the actions needed to achieve a set of goals. Finally, during the execute phase, the planned actions are executed, thereby changing the behavior of the managed element in the desired direction. All four functions have access to a shared knowledge base containing relevant symptoms to search for, behavioral policies, or goals. The knowledge can be inserted manually during design-time, retrieved and updated from an external source during run-time, e.g., from a central configuration server, or obtained by the autonomic system itself through online learning capabilities. Together with the knowledge base, the control loop of the autonomic element is often referred to as Monitor-Analyze-Plan-Execute loop with Knowledge (MAPE-K) [54].

The whole autonomic system typically consists of a collection of many autonomic elements. The system designers can set the abstraction that defines an element's boundary as it sees fit. For instance, an element can be a single CPU or software resource but also a large application service [49]. As such, the control exercised by the autonomic managers in each element can depict different granularities and functionalities. Also, it is possible to implement hierarchical control, where one orchestrating autonomic element manages not just itself but also exercises some degree of control over other elements [5].

2.3.2 *Organic Computing*

Another approach devised to tackle the increasing complexity in technical systems is Organic Computing (OC) [74]. The main concepts have been developed within the priority program SPP 1183 "Organic Computing" funded by the German Research Foundation (DFG). In contrast to AC, it does not focus on IT systems but on technical systems under computer control in general. The OC approach is also inspired by nature, where biological systems are inherently self-organized, and global objectives are pursued even without centralized global knowledge. Consequently, OC focuses on aspects of self-organization, even though the aim is not to

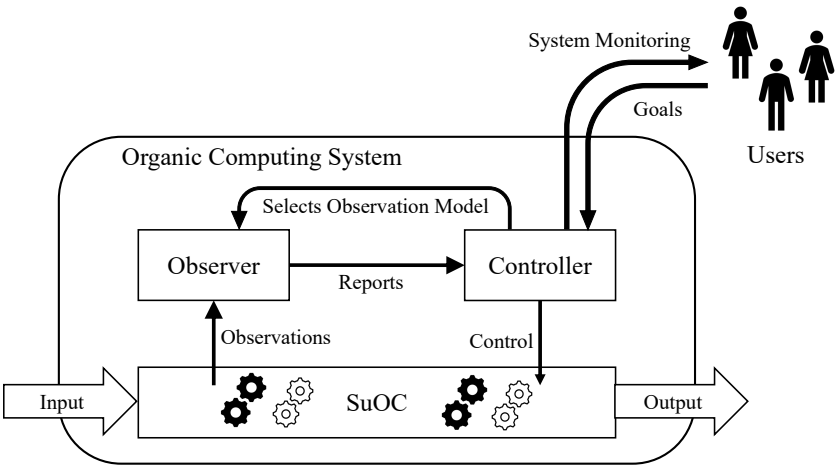


Figure 2.3: Observer/Controller architecture controlling a System under Observation and Control (SuOC) [72].

create fully autonomous systems. Instead, controlled self-organization is investigated, which allows the system to adapt to changing environments, goals, and constraints. At the same time, the system should not depict undesired behavior [89].

The main primitive employed in the OC concept is the observer/controller architecture shown in Figure 2.3. The System under Observation and Control (SuOC) is a productive (sub-)system that is, in principle, capable of functioning independently. Corrective actions are only taken when a deviation from the desired behavior is observed. The observer senses the system's state according to an observation model that is set by the controller and specifies which aspects are measured [72]. The gathered data is analyzed to generate a description of the current system state, which is then passed to the controller.

The controller generates an adequate reaction to the observed state and executes the control actions in the SuOC. Also, it can apply learning mechanisms to update its repertoire of behavioral strategies and send status information to higher-level systems or human operators. To achieve *controlled* self-organization, a human operator can update goals or even inject manual control actions.

Depicted in Figure 2.3 is a central observer/controller architecture, with one controller for the whole system. However, it is also possible to build decentralized and distributed architectures, where each subsystem is controlled by its own observer/controller architecture, and all subsystems cooperate. Another possibility are multi-level hierarchical architectures, where low-level controllers report to controllers on higher levels and receive their goals from them. This is a commonality with the approach of AC, as well as the control-loop pattern of the MAPE-K loop. However, in a MAPE-K loop, the knowledge is shared among all four functions. In contrast, in the observer/controller architecture, the knowledge is split into the observed information about the environment located in the observer and the procedural knowledge of the controller [72]. This loosely

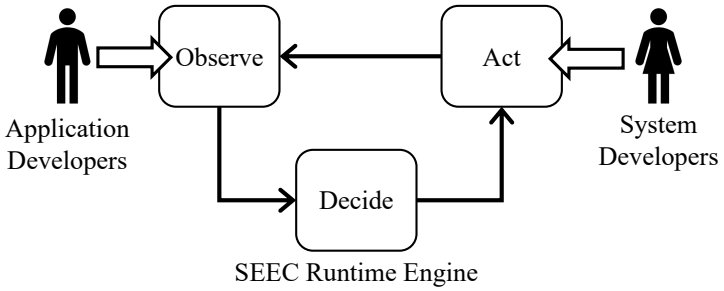


Figure 2.4: Observe-Decide-Act (ODA) loop of the SEEC framework with decoupled roles of application and system developers [43].

resembles the division between self-awareness and self-expression found in the notion of computational self-awareness of the EPICS project.

2.3.3 *Self-aware Computing Framework*

The SElf-awarE Computing (SEEC) framework proposed by Hoffmann et al. [43] has been specifically developed to manage the execution of applications on computing systems. During runtime, computing systems must often balance competing goals with different priorities, such as maximizing application performance, minimizing energy consumption, or adhering to performance or power constraints. SEEC aims to relieve application and system developers from fully understanding the complex interactions that arise when (multiple) applications are executed on a computing system with potentially competing objectives and interferences. SEEC achieves this by separating the concerns of both application and system developers. The application developers specify the application's goals and the feedback mechanism for measuring progress toward the goals. Optionally, application-level actions that affect the application performance can be specified. On the other hand, the system developers specify what system-level actions are available in the system's hard- and software and how they impact application performance. The SEEC engine then schedules actions at runtime to meet the specified goals efficiently while also considering environmental interferences.

The core of the SEEC framework is a decoupled ODA loop, as shown in Figure 2.4. The decoupling results in three distinct roles: Application developers, system developers, and SEEC runtime decision engine. In the observe step, the applications' current performance levels, progress toward their goals, and current system status are measured. The application developer's task is to specify the goals and how application performance is measured. Further, they must provide an implementation of the performance measurement that is accessible by the SEEC runtime engine. On the other side of the ODA loop, the system developer specifies what actions can be taken by the system. In addition, they must provide the estimated costs, e.g., the increase in power consumption, and benefits, e.g., expected speedups, associated with the actions. Also, it must be

specified for each action whether it affects only single applications or all applications running on the system and if conflicting actions exist. Even though the roles of application and system developers are distinct, as depicted in Figure 2.4, an application developer can also provide application-level actions to the act step, and a system developer can add system-level measurements to the observe step. However, the application developer does not need to consider the actions available in a specific system, while the system developer does not need to be aware of the runtime characteristics of the executed applications.

The arbitration between the running applications and the computing system is orchestrated by the SEEC runtime engine in the decide step, where appropriate actions are selected to optimize the execution of the applications in accordance with the specified goals. To achieve this, the decision engine of SEEC is designed with multiple layers of adaptation. An adaptive control system first determines the current gap or margin toward achieving the goals. The adaptive action scheduling engine then translates the gap or margin into a set of scheduled actions that meet the goals and minimize the costs. For this, the benefits and costs associated with each action are utilized. However, if the developers have specified them incorrectly, the control and action scheduling will show non-optimal behavior. To correct this, SEEC applies online learning to gather a more reliable estimate of the costs and benefits associated with each action. As such, the decision engine of SEEC employs a combination of adaptive control and machine learning. Overall, this reduces the effort for designing and managing computing systems, as application and system developers only specify aspects in their area of competence, while SEEC manages the system at runtime.

2.3.4 *Information Processing Factory*

The Information Processing Factory (IPF) project [29] is a collaboration between the UC Irvine (United States) and the TU Munich and TU Braunschweig (both located in Germany). As such, it is funded by both the National Science Foundation (NSF) and the German Research Foundation (DFG). The main aim of the IPF paradigm is to apply principles of computational self-awareness to the management of Multiprocessor System-on-Chips (MPSoCs), with the ultimate goal of reaching autonomous self-management. This is achieved through (i) self-reflection, which is defined as awareness of the MPSoC's own architecture, (ii) prediction of dynamic changes, and (iii) self-adaptation to both the predicted changes and unexpected situations [91].

The IPF design approach is inspired by the hierarchical organization of modern factories. Figure 2.5 shows the IPF organization for a hardware and software system that executes mixed-critical workloads [84]. The IPF consists of five layers:

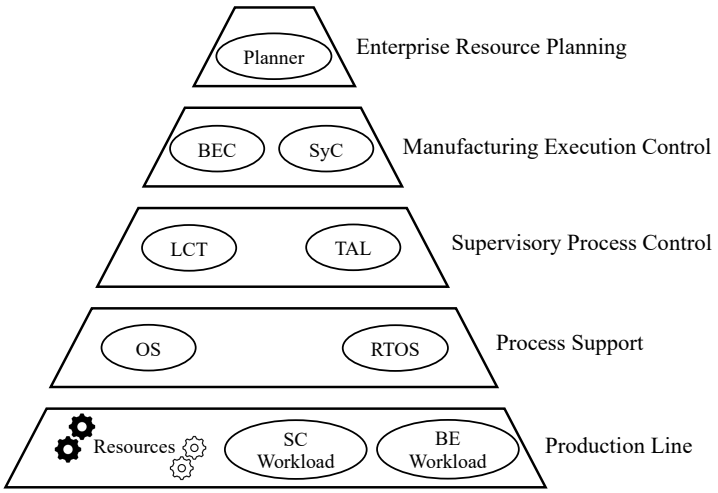


Figure 2.5: Information Processing Factory (IPF) organization for the execution of mixed-critical workloads [84].

- The lowest layer is the production line, which comprises the system's resources and executes the workload. The mixed-critical workload consists of best-effort tasks and safety-critical tasks, for which the compute resources must have a safety-critical mode with minimum performance guarantees. On the other hand, the configurable resources can enable their complete feature set when processing best-effort tasks that do not require deterministic performance levels.

- The second layer is the process support and consists of (real-time) Operating Systems (OSs) that support the workload execution by providing the execution environment.

- The computational self-awareness capabilities are added mostly in the third layer, termed the supervisory process control. The Trace Abstraction Layer (TAL) monitors the system for errors, while Learning Classifier Tables (LCTs) adjust the system configuration to optimize the workload execution, e.g., to minimize power consumption. An LCT is a variant of a Learning Classifier System (LCS) and is described in more detail in the case study in Chapter 6.

- As the fourth layer, the manufacturing execution control defines the operational region within the third layer can apply optimizations. The operational region for the best-effort tasks is determined by the Best-Effort Controller (BEC), while the System Controller (SyC) defines the safety-critical part of the operational region.

- In the fifth layer, the enterprise resource planning performs the long-term planning of the system. It determines a set of possible future operating regions, taking into account aspects such as the system's history, goals, constraints, and observed system degradation.

While the first version of the IPF aimed primarily at building dependable and autonomous MPSoCs, the second version [91] is proposed to expand the concept toward distributed systems of autonomous MPSoCs that can be applied in various application domains. One prominent application domain planned to be further investigated in the future is autonomous driving, where the MPSoCs in different cars form a distributed system.

2.3.5 *Relation to Computational Self-Awareness*

All presented approaches have in common that they tackle the increasing complexity of computing systems. The systems are better equipped to cope with unknown situations or interference during runtime by introducing adaptivity. This also lifts the burden on system designers and developers, as many design-time decisions are moved to the runtime. Since the runtime decisions are under the system's purview, system operators are also relieved. As such, the presented approaches mainly differ in which of these aspects the focus lies on. OC is the most general concept, as it is designed to apply to all technical systems under any form of computer control. Also, it encompasses any behavior that results in increased self-organization, which allows for various implementation opportunities. The AC initiative is focused on the management of large-scale IT systems and thus has a narrower field of envisioned application. Increasing the level of autonomy of IT systems is not primarily motivated by unforeseen situations that might arise or complex operational environments but by the increasing demand of system operators, which can slow down the growing dissemination of computing systems. SEEC is even more specific and focuses on managing a single computing platform that executes applications provided by developers. It lifts the burden on developers by decoupling the roles of application and system developers. Both application and system developers only specify aspects in their area of competence, and the SEEC runtime engine then manages the execution at runtime. Similar to SEEC, the IPF project focuses on managing computing systems, specifically MPSoCs. However, it has a broader range of envisioned application domains due to the explicit consideration of distributed systems.

The concept of computational self-awareness, as developed in the EPICS project, is similarly general as OC and can also be applied to general technical systems under computer control. In fact, computational self-awareness even applies to a larger number of systems, as basic stimulus-awareness is sufficient to qualify as self-aware and self-expressive behavior is not necessarily required. However, even if stimulus-awareness is used to implement simple reactive behavior, this does not have to represent a control loop, as it is the case for a MAPE-K loop, an observer/controller architecture, or an ODA loop. Still, to realize more complex adaptive behavior, the interactions of a self-aware system will often represent a control loop. The control loops defined by MAPE-K, the

observer/controller architecture, and ODA are very similar and mainly differ in the granularity with which the functionality is split. A similar approach can be seen in the reference architecture for computational self-awareness, where self-awareness and self-expression are split. Within the self-awareness engine, capabilities are grouped into different layers according to the type of knowledge and information processing. In terms of knowledge processing, the other presented concepts are less detailed.

The reference architecture for self-aware computing systems can incorporate self-awareness capabilities inherently present in the system by design, such as existing knowledge and information processing. In contrast, the control loops of AC and OC are added on top of a system. As such, the concept of computational self-awareness assumes that an existing system is internally modified to depict (improved) self-awareness capabilities, while the other concepts mainly intend to extend the system externally. The last significant difference concerns the focus of the concepts. The reference architecture for computational self-awareness extensively categorizes all processes associated with knowledge gathering by distinguishing private and public self-awareness, different layers of self-awareness capabilities, and even the explicit consideration of meta-self-awareness. On the other hand, the outlined related concepts focus more on the resulting behavior of the system. Overall, computational self-awareness is concerned mainly with how knowledge is generated, while the other approaches have a clear focus on behavioral capabilities, such as self-configuration or self-adaptation. However, systems that are designed in the realm of one of the presented concepts can, in many cases, also be categorized in one of the other concepts. Therefore, research on computing systems that depict some form of adaptive behavior is often not associated with precisely one of the presented concepts but is conducted under umbrella terms such as self-* properties, which is a term that consolidates all kinds of self-directed behaviors, e.g., self-organization, self-healing, or self-configuration.

2.4 APPLICATIONS OF SELF-* COMPUTING

To show that computational self-awareness and related concepts are not purely theoretical but can be filled with life, this section presents a selection of research works that have investigated systems with self-* properties . Naturally, this overview constitutes just a small insight into existing research, as whole conferences and journals are focusing on design methodologies and case studies of autonomous computing systems. Interested readers are therefore referred to the international conference on Autonomic Computing and Self-Organizing Systems (ACSOS), its predecessors, i.e., the International Conference on Autonomic Computing (ICAC) and the international conference on Self-Adaptive and Self-Organizing Systems (SASO), or related venues. However, in many cases, examples of self-* systems can be found in other publications venues as well, even if the venues do not have a clear focus on such topics. Three

works on self-aware computing platforms have been selected for this overview, along with a case study on a self-aware camera network as an example of collective self-awareness. In addition, two approaches focusing on traffic management are presented to highlight that computational self-awareness is not restricted to bare computing systems.

Agne et al. [1] use the reference architecture for computational self-awareness to build a self-aware heterogeneous compute node. The node comprises a CPU and a programmable logic fabric, i.e., a Field Programmable Gate Array (FPGA). The tasks responsible for implementing the self-awareness capabilities run on the CPU, while the compute tasks are executed on the FPGA. Using partial reconfiguration, the system can decide which portion of the FPGA is allocated to which task, i.e., how many threads are allocated to each task. Thereby, it can control how much workload per unit time is processed by each task. The system is experimentally evaluated using two applications with different priorities. One application has the highest priority and should not miss any input packets, which can happen if its input FIFO overflows. The second application is considered less critical but is assumed to have an infinite stream of inputs and should maximize the amount of processed data. Two static allocation strategies are compared alongside a meta-strategy that, at runtime, decides which of the two strategies is appropriate. This results in a more adaptive behavior than with any of the static strategies alone. Since the meta-strategy does algorithm selection, it can be considered a form of meta-self-awareness. Overall, this example of a self-aware compute node shows that implementing capabilities of computational self-awareness does not necessarily require online learning.

To specifically support the SEEC framework, Hoffmann et al. [44] have designed the Angstrom processor. The observe step of the ODA loop is supported by the processor with several new sensors, e.g., for performance counters, temperature, battery level, and energy consumption. Unlike in traditional processors, the sensors are designed to be read efficiently at a large scale. The act step of the ODA loop is extended with actions commonly not present in off-the-shelf processors, e.g., frequency and voltage scaling for every core and runtime reconfiguration of caches to optimize power by disabling unnecessary parts of the cache. Inter-core adaptations are available as well to tailor the on-chip network and the used cache coherence protocols to the executed applications. A simulation environment is used to evaluate the Angstrom processor. Using the heartbeat API [42], applications register a desired level of performance. The SEEC runtime engine then tries to reach the performance level with minimal energy consumption by employing adaptive control and machine learning. As such, the Angstrom processor depicts no self-* properties by itself but supports the design of self-* systems by supporting more observations and actions. This enables the runtime engine to make more informed decisions that are better suited to the current situation.

Burger et al. [16] design self-aware sensor nodes that can distribute the computations in the sensor network. Each sensor node has a resource-

constrained System-on-Chip (SoC) containing a CPU and an FPGA to process sensor information. When an information-processing task arrives, each node can process it locally or offload it to another node. When processing a task locally, the current FPGA configuration might not match the task. Since the reconfiguration of the FPGA can impose a significant overhead, the node can decide to place the task in a local queue. When processing the tasks from the queue in a batched manner, unnecessary reconfigurations can be avoided if tasks of the same type are processed sequentially. The decision of which action is selected is based on the current battery level of the node, the number of batched jobs, and the current FPGA configuration. The decision-making is done via tabular Q-learning, a Reinforcement Learning (RL) technique. The goal is to process as many jobs as possible in the sensor network without depleting the batteries. In a simulation, it is shown that Q-learning indeed learns desirable node behavior and can adapt to network failures.

Explicit cooperation in a network is employed by Rinner et al. [86], who propose the design of a self-aware smart camera network for object tracking. The capabilities of the smart cameras can be categorized into the self-awareness layers of the reference architecture. The object tracking itself falls into the category of stimulus-awareness, as it is based solely on the current camera feed. The handover of an object to a different camera occurs when the object has left the camera's point of view or when the object has moved and another camera has a better angle. The handover decision is made via an auctioning approach. To avoid that the handover auction always needs to be broadcasted to all cameras in the network, each camera locally learns the network topology to identify neighboring cameras. Since cameras can join or leave the network at runtime, each camera must have a notion of the past to identify outdated network topologies, which is why topology learning classifies as time-awareness. Each camera can follow different strategies for object handover that determine when a handover should take place and which cameras can participate in the auction. Online learning, i.e., a multi-armed bandit problem solver, is employed to select the most appropriate strategy at runtime. This strategy selection constitutes a form of meta-self-awareness.

As part of the OC initiative, Prothmann et al. [82] have employed the observer/controller architecture for traffic control. The SuOC is a standard traffic light controller with configurable parameters that controls the traffic lights in an intersection. Instead of the original observer/controller architecture, a layered approach is employed. The first layer is responsible for selecting the parameters of the traffic light controller. The observer component monitors the current traffic flows into the intersection and summarizes this into a vector representing the current traffic situation. The vector is then fed into a LCS, which acts as controller component. The LCS searches its rulebase and selects a parameterization of the traffic light controller that is appropriate for the current situation. Since rule generation through evolutionary learning is a stochastic process that frequently generates inadequate rules, the generation of rules does

not occur directly in the LCS. Instead, new rules are generated in the second layer of the hierarchy. New rules are still generated with evolutionary learning but validated in a simulation afterward. Only if a rule performs sufficiently well in the simulation it is inserted into the LCS. This approach assures that all rules in the LCS are helpful or at least do not have a considerable detrimental impact, thereby assuring that the operational performance of the traffic light controller is not negatively impacted during runtime. Thus, this approach is an example of *controlled* self-organization – the prevalent feature of OC. Simulation results show that the intersection achieves better average delay times under the control of the layered observer/controller architecture than with static control strategies.

Also in the field of traffic control, Lesch et al. [63] investigated a situation-aware selection of platooning strategies and their parameterization. A platoon encompasses multiple vehicles driving behind each other at low distances to save fuel and better utilize the road infrastructure. The control approach is layered, as on the first layer the platooning strategy is managing the platoon, while the layer above employs a meta-optimization strategy to select the platooning strategy and tailor its configuration parameters. As in some of the other examples, this algorithm selection represents a form of meta-self-awareness. The meta-optimization layer is designed as a MAPE-K loop, where the monitor phase gathers necessary information, such as the number and properties of the platoon vehicles. The analyze phase calculates the current performance with respect to the current objectives. In the plan phase, it is determined if the current platooning strategy should be switched and the configuration parameters are optimized, even if the strategy remains unchanged. For optimization of the parameters, several learning techniques are compared, among them a Genetic Algorithm (GA) and simulated annealing. The execute phase then changes the platooning strategy or the configuration parameters accordingly. Relevant statistics are stored in the knowledge component to improve decision-making in the future. A simulation shows that the different platooning strategies are suited for different objectives and that the optimal parameter configuration of each strategy depends on the current traffic scenario, highlighting the need for a situation-aware selection of strategy and parameterization.

3

THE LEARNING CLASSIFIER SYSTEM XCS

Learning Classifier Systems (LCSs) are a class of techniques from the domain of Rule-based Machine Learning (RBML) that aim at learning a ruleset consisting of several if-then rules. Together, the rules cover the whole input space and propose for each possible input the correct action or class [106]. The ruleset is learned via means of Evolutionary Computation (EC), typically with a Genetic Algorithm (GA). In Michigan-style LCSs, the GA operates on individual rules ("classifiers") with an incremental learning scheme, i.e., sample by sample. In contrast, the GA in Pittsburgh-style LCSs operates on whole rulesets and employs batch learning. As such, Michigan-style LCSs are often employed for Reinforcement Learning (RL) applications because they are considered to be more flexible, and Pittsburg-style LCSs are being predominantly used for supervised learning [106].

Most popular nowadays is XCS [110], which is a Michigan-style LCS that interacts with the problem environment according to the RL paradigm as shown in Figure 3.1. XCS senses the current state of the environment and uses the resulting sensory input to search its ruleset, also termed classifier population, for matching rules. The rules then determine the action that is executed in the environment. As feedback, the environment returns a reward value, with large rewards indicating desirable outcomes and low rewards indicating undesirable effects. In single-step problems, environmental inputs are independent of each other, and the reward solely depends on the current state of the environment and the action taken. This is not the case in multi-step environments, where a series of actions is required to solve a problem instance and maximize the overall reward. The goal in single-step environments is to maximize the immediate reward, while in multi-step environments, the rewards received in future states must also be considered. A reinforcement learner, such as XCS, is then said to maximize the long-term value, also termed payoff, of the observed state [99].

The distinguishing feature of XCS is that its internal learning mechanism aims to accurately predict the reward that will be received instead of simply maximizing it. To this end, the fitness that is associated with each rule and guides the steady-state niche GA of XCS is based on the accuracy of the reward prediction. The fitness cannot be computed directly, as it is done in standalone GAs, but is iteratively learned via RL.

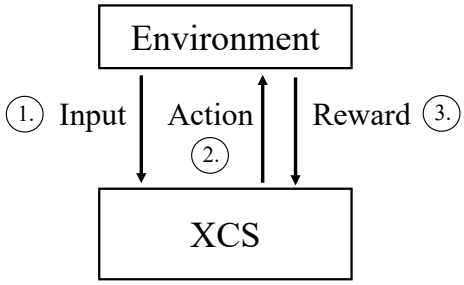


Figure 3.1: The interaction of XCS with the problem environment.

In contrast to traditional applications of GAs, XCS employs the GA not to maximize a function but to generalize its rules adequately.

Since its initial development, several different variants of XCS with various, sometimes subtle, implementational differences have been proposed. Hence, *the* standard XCS does not exist. The chapter continues with Section 3.1 by presenting the algorithmic details of the most common version of XCS as described in [17], which forms the basis of most experimental evaluations discussed in this thesis. Based on its algorithmic structure, Section 3.2 discusses the working mechanisms of XCS and how it evolves a ruleset that covers all environmental states with as few but maximally generalized rules. Section 3.3 then gives a short overview of some of the most popular extensions that have been proposed to XCS to make it applicable to a broader range of problem environments. Lastly, Section 3.4 discusses why XCS can be considered a good fit for implementing computational self-awareness.

3.1 ALGORITHMIC DESCRIPTION OF XCS

The main components of XCS are shown in Figure 3.2. When interacting with the environment, XCS receives a sensory input $\sigma(t) \in \{0,1\}^L$ encoded as a bitstring of length L, decides for an action $a(t) \in \{a_1, \ldots, a_n\}$ and receives a scalar reward r from the environment after $a(t)$ has been executed. The reward is then used to guide the learning process inside XCS. The core of XCS is a population $[P]$ of classifiers, which can be interpreted as if-then rules. Each classifier cl consists of a condition $C \in \{0,1,\#\}^L$, which defines for which inputs the classifier applies. The don't cares '#' allow XCS to generalize and create classifiers that match multiple inputs. Each classifier proposes exactly one action $a \in \{a_1, \ldots, a_n\}$ for all situations that match its condition. In addition, the classifier keeps an estimate of the payoff (or reward) p that is expected to be received when the classifier matches a situation and a is executed. The parameter ϵ represents an estimate of the prediction error associated with the payoff prediction and is used to calculate the fitness F, which guides the GA to evolve accurate and general classifiers. Further, each classifier has an experience exp, denoting how often the classifier matched a situation and its action was executed, an estimation of the average action set size as, and a

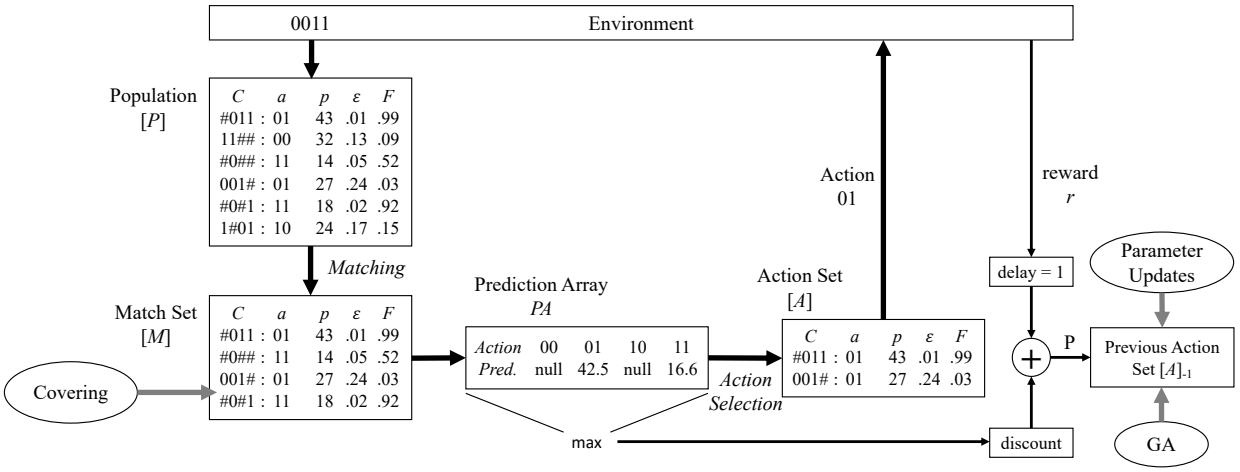


Figure 3.2: Illustration of XCS, taken from [112], with a 4-bit input, available actions $a \in \{00, 01, 10, 11\}$ and the minimum number of actions θ_{mna} set to 2. The environment provides the sensory input 0011, and XCS selects action 01 for execution. Within single-step environments, the discounting mechanism and the previous action set $[A]_{-1}$ are not necessary, as the GA and the parameter update take place on the current action set $[A]$ using the immediate reward r.

numerosity n. For reasons of computational efficiency, identical classifiers do not occur multiple times in the population but are summarized to one classifier with $n > 1$. The numerosity plays an important role during learning, as the population $[P]$ has a maximum size N, specified by the system designer, such that $\sum_{cl \in [P]} cl.n \leq N$ is always fulfilled. Typically, the number of distinct classifiers in the population is well below N, as most classifiers have a numerosity n higher than one, especially once XCS has learned adequate generalizations.

Upon receiving an input $\sigma(t)$, XCS executes four steps:

1. Identification of matching classifiers to form the match set $[M]$.
2. Action selection to form the action set $[A]$.
3. Execution of the action. The received reward is used to update the parameters of all classifiers in $[A]$ ($[A]_{-1}$ in multi-step problems).
4. Applying the GA to $[A]$ ($[A]_{-1}$ in multi-step problems).

Steps 1 and 2 are often termed the *performance component* of XCS, while steps 3 and 4 form the *reinforcement component* and *discovery component*, respectively [20]. Overall, XCS aims to evolve a population of classifiers that are accurate and maximally general, i.e., have the maximal number of don't cares in their conditions while still exhibiting a low prediction error ϵ. If successful, this leads to a population of a minimal size that can predict for every situation the best-suited action. The goal of evolving such a population is inherent to each of the four steps.

1) Creation of the Match Set. The whole population is searched for classifiers whose condition C matches the current input $\sigma(t)$. Matching

classifiers are added to the match set $[M]$. By default, it is required that all available actions are present in $[M]$, but this can be changed by setting the configurable hyperparameter θ_{mna}, which specifies the minimum number of different actions that must be present in $[M]$. In case not enough actions are present, a new matching classifier is randomly created via a covering procedure: The condition C is selected to match the current input $\sigma(t)$, with don't cares introduced according to the configurable probability $P_{\#}$. As action a, one of the actions not yet present in $[M]$ is randomly selected, and the remaining parameters are initialized with default values. This covering procedure repeats until the match set $[M]$ contains at least θ_{mna} different actions.

2) Action Selection. Once all matching classifiers have been identified, one of their actions must be selected for execution. To choose actions that maximize the payoff, XCS builds the prediction array PA, which holds for every action the fitness-weighted average of the payoff predictions of all classifiers in $[M]$ that propose this action. Since the fitness of a classifier is based on its prediction accuracy, more accurate payoff predictions are weighted higher. Several action selection strategies are possible, but the most commonly used is an ϵ-greedy strategy, in which either pure exploitation, i.e., selecting the action with the highest prediction, or pure exploration, i.e., random selection, is conducted. After an action has been selected, the action set $[A]$ is formed with all classifiers of $[M]$ that propose the selected action.

3) Parameter Update. After the action has been executed in the environment, the parameters of the classifiers in $[A]$ are updated. First, the experience exp of each classifier is incremented by one, followed by the updates of the payoff estimate p and the prediction error ϵ. In single-step environments, XCS uses the immediate reward r as the payoff to update the classifiers in $[A]$, i.e., $P = r$. In multi-step environments, the update takes place on the previous action set $[A]_{-1}$ and calculates the payoff P as the sum of the previous immediate reward and the discounted maximum of the current prediction array, i.e., $P = r_{-1} + \gamma \cdot max(PA)$. This ensures that immediate rewards received at later points in time are propagated to preceding classifiers, which have paved the way for receiving this reward.

The prediction p and error estimate ϵ of each classifier cl are updated according to the Widrow-Hoff delta rule, as shown in Equation 3.1 and Equation 3.2, respectively. The average action set size as is updated in the same manner as shown in Equation 3.3.

$$cl.p = \begin{cases} cl.p + \frac{P - cl.p}{cl.exp} & \text{if } cl.exp < \frac{1}{\beta} \\ cl.p + \beta \cdot (P - cl.p) & \text{else} \end{cases} \tag{3.1}$$

$$cl.\epsilon = \begin{cases} cl.\epsilon + \frac{|P - cl.p| - cl.\epsilon}{cl.exp} & \text{if } cl.exp < \frac{1}{\beta} \\ cl.\epsilon + \beta \cdot (|P - cl.p| - \epsilon) & \text{else} \end{cases} \tag{3.2}$$

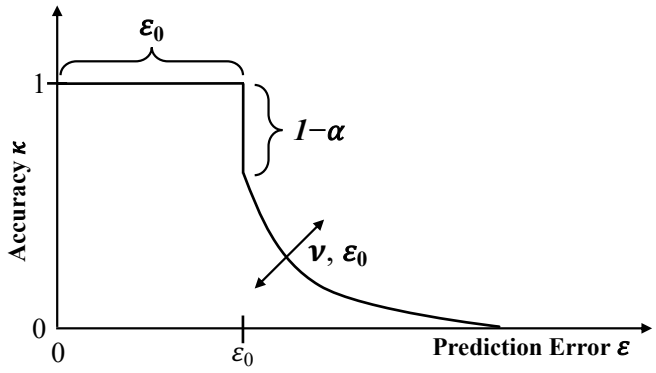


Figure 3.3: Classifier accuracy κ in relation to the classifier's prediction error ϵ. Figure adapted from [20].

$$cl.as = \begin{cases} cl.as + \frac{\sum_{c\in[A]} c.n - cl.as}{cl.exp} & \text{if } cl.exp < \frac{1}{\beta} \\ cl.as + \beta \cdot \left(\sum_{c\in[A]} c.n - cl.as\right) & \text{else} \end{cases} \tag{3.3}$$

The configurable learning rate β has a value between 0 and 1 and controls the magnitude of the parameter change. For a quicker initial conversion, β is exchanged with $\frac{1}{cl.exp}$ as long as the classifier is considered inexperienced.

To update the fitness F, two steps are required. First, the accuracy κ of each classifier $cl \in [A]$ is determined according to Equation 3.4. If the estimated prediction error ϵ lies below the configurable error threshold ϵ_0, the accuracy is assigned the highest value of 1. Otherwise, the accuracy decreases exponentially, with the hyperparameters $\alpha \in [0,1]$ and $\nu \geq 1$ controlling the behavior of the decrease, as visualized in Figure 3.3. The fitness F represents the relative accuracy and is calculated as the accuracy of the classifier in relation to the sum of the accuracies of all classifiers in $[A]$, as shown in Equation 3.5. This fitness sharing, in which the fitness of the environmental niche is distributed among all matching classifiers, is one force pushing the evolutionary process in XCS towards an even allocation of classifiers to different environmental niches [106].

$$\kappa(cl) = \begin{cases} 1 & \text{if } cl.\epsilon < \epsilon_0 \\ \alpha \cdot \left(\frac{cl.\epsilon}{\epsilon_0}\right)^{-\nu} & \text{else} \end{cases} \tag{3.4}$$

$$cl.F = cl.F + \beta \cdot \left(\frac{\kappa(cl) \cdot cl.n}{\sum_{c\in[A]} \kappa(c) \cdot c.n} - cl.F\right) \tag{3.5}$$

After all parameters have been updated, *action set subsumption* takes place if enabled by the system designer. Subsumption aims to remove overly specific classifiers in favor of more general classifiers representing

the same knowledge. This condenses the classifier population and enables XCS to solve the problem with a smaller population. The action set $[A]$ is searched for the most general classifier $cl_{general}$, i.e., the classifier with most don't cares in its condition, that is sufficiently experienced and accurate. A classifier is considered sufficiently experienced and accurate if its experience exp is higher than the configurable hyperparameter θ_{sub} and its prediction error estimate ϵ is below ϵ_0. All classifiers in $[A]$ that cover only a subset of the space covered by the condition of $cl_{general}$ are removed from the classifier population $[P]$, and their numerosity is added to the numerosity of $cl_{general}$. Action set subsumption is a strong subsumption mechanism that can quickly reduce the population size [20]. However, in practice, it often turns out to be too powerful, which is why it is typically not enabled.

4) Genetic Algorithm. During exploration iterations, new classifiers are generated by the GA. The GA is executed on $[A]$, or on $[A]_{-1}$ in multi-step environments, if the average time period since the classifiers in the action set participated in the GA is greater than the configurable threshold θ_{GA}. If so, two classifiers are selected from $[A]$ via a roulette-wheel selection based on the fitness F. In this roulette-wheel selection, a classifier is randomly selected with a probability proportional to its fitness – akin to spinning a roulette-wheel with all classifiers from $[A]$ on it, where the size of each classifier's pocket is proportional to its fitness. Hence, classifiers with high fitness, corresponding to a high level of accuracy, are more likely to be selected for reproduction. To create two offspring from the selected classifiers, two-point crossover is applied to the classifiers' conditions with a probability χ. In two-point crossover, two cutpoints in the condition are randomly chosen, and the offspring's conditions are formed by recombination of the resulting sections of the parents' conditions. Afterward, each entry of the condition of the offspring is mutated with probability μ, while ensuring that the condition still matches the current input $\sigma(t)$. Further, the action a is mutated with probability μ as well. If enabled by the system designer, *GA subsumption* takes place before inserting both offspring into the population. When the parent classifiers are sufficiently experienced and accurate, as determined by the thresholds θ_{sub} and ϵ_0, and the offspring covers only a subset of one of its parents' conditions and proposes the same action, the parent is said to subsume its offspring. Then, it is not inserted into the population, as it does not represent new knowledge. Instead, the numerosity n of the respective parent is incremented by one.

Whenever a new classifier is added to the population and the maximum population size N is exceeded, a classifier is selected for deletion from the population $[P]$ with a roulette-wheel selection. The deletion vote of each classifier is calculated according to Equation 3.6 and is proportional to its action set size estimate as and numerosity n. If it has sufficient experience, as determined by θ_{del}, and a fitness F lower than a fraction δ of the average fitness of the classifier population, its deletion vote is further increased inversely with its fitness. Hence, classifiers are favored

Table 3.1: Overview of XCS' hyperparameters and common default values.

Parameter	Description	Common value
N	Maximum population size	Problem-dependent
θ_{mna}	Min. number of actions in $[M]$	All possible actions
$P_{\#}$	Covering don't care probability	$0.3 - 0.7$
p_I	Initial prediction of classifier	1 % of reward range
ϵ_I	Initial prediction error	0
F_I	Initial fitness	0.01
γ	Discount factor (multi-step)	$0.7 - 0.95$
β	Learning rate	0.2
ϵ_0	Error threshold	1 % of reward range
α	Shape of accuracy function	0.1
ν	Shape of accuracy function	5
$[A]$ subs.	Enable action set subsumption	False
θ_{sub}	Subsumption threshold	20
θ_{GA}	GA activation threshold	25
χ	Crossover probability	0.8
μ	Mutation probability	0.04
GA subs.	Enable GA subsumption	True
θ_{del}	Experience threshold deletion	20
δ	Fitness threshold deletion	0.1

for deletion if they have low fitness, high numerosity, and frequently occur in large action sets. The aim is not only to remove inaccurate classifiers with low fitness but also to balance the classifier population towards covering each environmental niche equally, which is achieved by considering the action set size as. If a classifier with a numerosity $n > 1$ is selected for deletion, the classifier is not removed from the population, but its numerosity is decremented by one.

$$\text{vote}(cl) = \begin{cases} cl.as * cl.n * \frac{avgFit}{\frac{cl.F}{cl.n}} & \text{if } cl.exp > \theta_{del} \text{ and} \\ & \quad \frac{cl.F}{cl.n} < \delta \cdot avgFit \\ cl.as * cl.n & \text{else} \end{cases} \tag{3.6}$$

Throughout all described steps, the behavior of XCS is affected by a plethora of configurable hyperparameters, which are summarized in Table 3.1 along with common default values. Each hyperparameter can, and sometimes must, be tailored to the problem environment. The given values merely represent a starting point often used in the research literature.

3.2 THE WORKING MECHANISM OF XCS

While the previous section has described *how* XCS works algorithmically, it is still open *why* it works and learns to solve a problem with a population of generalized classifiers. When XCS was proposed and became the most popular LCS shortly after, the interactions between the different components inside XCS have not been investigated systematically and were not fully understood. Instead, discussions of XCS research have been based primarily on intuition – and still are nowadays. For instance, Wilson's generalization hypothesis that "[...] it appeared that the interaction of accuracy-based fitness and the use of a niche GA could result in evolutionary pressure toward classifiers that would be not only accurate, but both accurate and maximally general" [110] remained a hypothesis for several years that was based on intuition and backed up only by loose experimental results. This is astonishing, as the characteristics of the classifier population evolved by XCS are the most vital aspect of XCS' learning mechanism.

The goal of this section is not to give an exhaustive overview of existing theoretical research on the working mechanism of XCS. Instead, it aims to foster the understanding of XCS beyond its algorithmic properties. As such, this section is especially suited for readers who still need to become familiar with XCS, as fully understanding its inner mechanics takes considerable personal experience and reviews of research literature. First, Wilson's generalization hypothesis is revisited, and theoretical insights on the generalization behavior of XCS are summarized. Afterward, strength-based and accuracy-based fitness is compared. The main difference between XCS and preceding LCSs is that XCS uses prediction accuracy as the basis for classifier fitness. A comparison between strength- and accuracy-based fitness in LCSs not only justifies why XCS became the most popular LCS but also gives insights into the strengths and weaknesses of XCS.

3.2.1 *Generalization Pressure in XCS*

Even though XCS was invented in 1995, it took until 2004 that Butz et al. [20] presented the first extensive theoretical analysis of the learning mechanism of XCS and investigated the cause for its tendency to evolve accurate classifiers that are maximally general. The generality of a classifier describes how many input states a classifier matches to and is defined as the fraction of don't care symbols # in the classifier condition. A high generality is favorable, as it allows XCS to cover the whole observation space of the environment with a smaller classifier population. The basis of the theoretical analysis is formed by different evolutionary pressures in XCS that influence the generality of the evolved classifiers [20].

Set Pressure. The set pressure is the basis for Wilson's generalization hypothesis and results from how classifiers in the population are reproduced and deleted. The creation of new classifiers takes place by

applying the niche-based GA in the action set [A], which implies that offspring classifiers also match the current environmental input. Classifier deletion, on the other hand, takes place globally on the whole classifier population [P]. Since more general classifiers will match more frequently to the observed inputs, they will also occur more frequently in the action set [A]. They will thus have more opportunities for reproduction. As a result, the offspring created from those classifiers will likely also be generalized above average. Once the offspring classifiers have been generated, classifier deletion takes place to make room in the population for the newly created classifiers. The global deletion acts on the whole population [P] and does not consider the generality of classifiers. Hence, more general classifiers have a higher chance of reproduction but the same probability of getting deleted. This results in a tendency of XCS to evolve a population of maximally generalized classifiers. The described intuitive line of argumentation has been formally confirmed in [20]. For randomly initialized populations, they show that the generality of the classifiers in the action set is, on average, higher than those of the whole population.

Mutation Pressure. As part of the GA, mutation also influences the generality of the offspring classifiers. During the niche mutation typically employed in XCS, each don't care symbol # in the classifier's condition is mutated with probability μ to the specified value matching to the input, while each specified entry of the condition is mutated with probability μ to a don't care symbol #. As such, niche mutation pushes toward an equal distribution of don't care and specified symbols in the classifier condition, i.e., toward a generality of 0.5. However, the strength of the mutation pressure depends on the mutation probability μ, which is typically set to a low value, e.g., 0.04, and the frequency with which the GA is applied, which is influenced by the parameter θ_{GA} and the sampling distribution of the environmental inputs.

Deletion Pressure. Classifier deletion acts on the whole population. It has, neglecting the influence of classifier fitness, the tendency to delete classifiers frequently occurring in larger action sets, pushing the population toward an even allocation of classifiers to environmental niches. As such, the deletion pressure is independent of classifier generality, and classifiers selected for deletion will have, on average, the same generality as the population average. However, the effects of deletion on classifier generality must be considered if alternative deletion schemes are used [13, 50].

Subsumption Pressure. Even though their application is optional in XCS, both the GA subsumption and the action set subsumption mechanisms are likely the most apparent pressures toward general classifiers. After the GA has created offspring classifiers, it is checked if the offspring are subsumed by one of their parents, i.e., have a condition that is less general than that of a parent and is completely covered by it. In such a case, the offspring is not inserted into the population, but the numerosity of the more general parent is incremented. Action set subsumption acts

on the action set and is typically considered the stronger subsumption mechanism, as a general classifier can subsume multiple classifiers at once. However, a subsuming classifier must be sufficiently experienced and accurate for both subsumption mechanisms to act, as determined by θ_{sub} and ϵ_0. Hence, subsumption only acts once accurate classifiers have evolved and stops when classifiers become inaccurate. As such, subsumption pressure represents a pressure that pushes XCS to evolve maximally general classifiers that are still accurate. As noted by Butz et al. [20], classifier subsumption is not necessary for XCS to evolve accurate and maximally general classifiers but can lead to a substantial decrease in the population size once it kicks in during later stages of learning.

Fitness Pressure. So far, the influence of classifier fitness has not been considered, or, in the case of the subsumption pressure, only indirectly via the accuracy requirement on the subsuming classifier. In general, the fitness pressure pushes classifiers toward a high accuracy of the payoff prediction since classifiers with a high prediction accuracy get assigned a higher fitness. Typically, an inaccurate payoff prediction results from wrong generalization, which means that the fitness pressure pushes toward less generality as long as a classifier is inaccurate. During classifier deletion, experienced classifiers with a low fitness have a higher chance of being deleted, which means that wrongly generalized classifiers are driven out of the population. In addition, the GA uses the fitness value as the selection criterion, which means that classifiers with a high fitness value get more reproduction opportunities. Therefore, the fitness pressure can be seen as the counteracting force of the set pressure. Due to the set pressure, the classifiers in the action set have, on average, a higher generality than the other classifiers in the population. Nevertheless, due to the fitness pressure, the classifiers with high fitness that are not overly generalized are selected by the GA for reproduction.

Figure 3.4 shows the interaction of the different pressures. The set pressure pushes the classifier population toward maximum generality, while the mutation pressure pushes it to a fixed generality of 0.5. Classifier subsumption acts only on classifiers considered accurate and increases their generality. From the other direction, the fitness pressure reduces the generality of overly generalized classifiers with inaccurate payoff predictions. The deletion pressure is included both in the set pressure, as it operates on the whole population $[P]$, and the fitness pressure, since unfit classifiers have a higher chance of being deleted. Overall, these interactions result in XCS evolving a population of accurate but maximally generalized classifiers. Figure 3.4 also shows why the subsumption mechanisms are optional in XCS. The set pressure tends to increase the generality of the evolved classifiers, while the fitness pressure pushes against this, but only as long as the classifiers are considered inaccurate. This interaction alone should result in accurate and maximally generalized classifiers, but the convergence speed may be improved by subsumption. However, while the existence of the set pressure can be formally derived from the algorithmic properties of XCS rather easily,

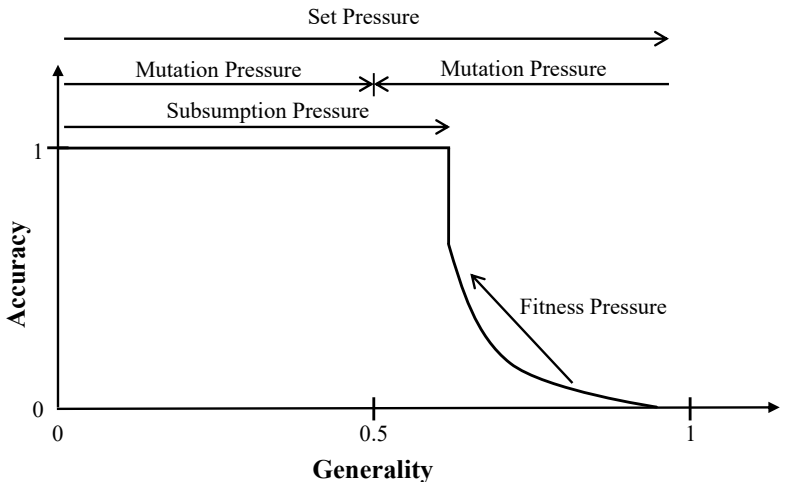


Figure 3.4: Interaction of the different evolutionary pressures in XCS. The set pressure pushes the classifier population towards maximal generality, while mutation pressure pushes towards a fixed generality of 0.5. Subsumption pressure is only applied to accurate classifiers and increases their generality. The fitness pressure applies to inaccurate classifiers and decreases their generality to make them more accurate. The deletion pressure is included in the set and the fitness pressure. Figure adapted from [20].

this is not the case for the fitness pressure which depends highly on the considered problem environment.

As general and problem-independent requirements for the fitness pressure to act, Butz et al. [20] have identified two conditions, termed challenges. The *covering challenge* states that the classifier fitness on which the GA acts must be meaningful, while the *schema challenge* ensures that the GA results in an effective pressure toward accurate classifiers with high fitness.

The *covering challenge* is concerned with setting the parameter configuration of XCS in a way that the GA acts on meaningful fitness values. Since classifier fitness in XCS is determined by interaction with the environment and cannot be computed as in standalone GAs, classifiers must be applied in the environment multiple times until their fitness values become meaningful. Initially, the classifier population is empty, and classifiers are randomly generated via covering to match the inserted inputs. After further iterations, the GA will begin to act in the action sets. At this point, the classifiers must be sufficiently experienced so that the GA operates on meaningful fitness values. Typically, covering only acts at the beginning of learning until the randomly created classifier population covers the whole input space of the environment. After this, the GA refines the classifiers based on their fitness. It can be the case, however, that XCS never stops covering because the classifier population has reached its maximum size but still does not cover the whole input space. In such a case, covering is forced to create additional classifiers for so-far unseen inputs but cannot insert them into the population without

deleting classifiers. Since the classifiers in the population are still inexperienced and their parameters, such as the prediction error and the action set size, are rather meaningless, deletion will essentially be random. XCS is stuck in a cover-delete cycle, and classifiers never get the chance to become experienced enough that their fitness values become meaningful. As a result, the GA operates on classifiers with meaningless fitness, and fitness pressure is not emerging. The covering challenge can be overcome by setting the don't care probability $P_\#$ of the covering operator and the maximum population size N to appropriate values. A larger population accommodates more classifiers and thus increases the chance that the population covers a sufficiently large amount of the input space. In contrast, a higher value of $P_\#$ increases the generality of classifiers created by covering, meaning each classifier covers a larger area of the input space.

When the covering challenge is met, it is ensured that the classifiers in the population sufficiently cover the input space and have been evaluated often enough that the fitness values are representative of the classifier accuracy. As the second requirement for the emergence of fitness pressure, the *schema challenge* must be met. This assures that the GA has enough sufficiently accurate classifiers to work with. Only then can it evolve offspring classifiers with even higher accuracy, resulting in pressure toward higher fitness. This can be achieved by setting $P_\#$ to a small enough value to ensure that classifiers in the population have enough specified bits to avoid being inaccurate due to overgeneralization. For instance, consider the case that XCS has met the covering challenge by having a higher value of $P_\#$. The initial population will consist of highly generalized classifiers that cover large areas of the input space due to the high number of don't cares in their conditions. Because of this, they will likely be too generalized and have inaccurate payoff predictions.[1] If all classifiers are similarly inaccurate, they will have similar fitness values, and the GA will essentially select random classifiers for reproduction. This makes the classifier evolution inside XCS undirected, preventing the emergence of fitness pressure toward more accurate classifiers.

Meeting both the covering and the schema challenge are requirements for the existence of the fitness pressure but do not guarantee the latter. Overall, the covering challenge can be met by setting the don't care probability $P_\#$ of the covering operator and the maximum population size N to sufficiently large values. In contrast, the schema challenge requires sufficiently small values of $P_\#$. What is "sufficient" is highly problem-dependent, as is the fitness pressure as a whole. For instance, adequate parameter configurations for meeting the covering and the schema challenges depend on the sampling distribution of input states, the reward function, and the number of generalization opportunities offered by the problem environment. Butz et al. [20] have experimentally validated the existence of both challenges in several Boolean multiplexer environments

1 The appropriate degree of generalization that still leads to accurate payoff predictions is highly problem-dependent, but every non-trivial problem environment can suffer from too much generalization at some point.

and have formally derived suitable parameter values. However, since their approach of determining suitable parameter values requires knowing the condition structure of the accurate and maximally general classifiers apriori, it is not applicable to non-artificial and complex learning problems. In practice, parameters are typically set to common default values, e.g., $P_\# = 0.5$, or are determined based on intuition and trial-and-error experiments.

3.2.2 *Classifier Fitness: Strength vs. Accuracy*

The early classifier systems based the fitness of a classifier on its strength, i.e., the magnitude of its payoff prediction. Hence, classifiers that lead to a higher payoff get assigned a higher fitness, which seems like a natural choice to maximize the reward received in an environment. XCS was the first LCS that used the accuracy of the payoff prediction as the basis for classifier fitness, which is why it is commonly referred to as an accuracy-based classifier system. Shortly after the concept of accuracy-based fitness was proposed, the LCS research community turned its focus more and more toward XCS, and strength-based LCSs received a decreasing amount of attention. XCS, and other variants of accuracy-based classifier systems derived from it, are still the most popular type of LCSs nowadays [41, 80]. This subsection outlines differences between strength-based classifier systems and XCS to highlight the reasons for the success of accuracy-based classifier systems and foster the understanding of the strengths and weaknesses of XCS. Even though the fitness evaluation is a fundamental aspect of an LCS, where the differences between strength and accuracy as fitness metric can be discussed and evaluated to the extent of a whole PhD thesis [53], this subsection restricts the discussion to the core ideas as presented by Kovacs in [51].

Classifier Allocation. In strength-based LCSs, classifier creation is greedy since the GA favors the classifiers for reproduction that lead to the highest rewards. If the environment has a biased reward function, which means that selecting the best action in different states leads to rewards of different magnitudes, the input space of the environment may be unevenly covered by a strength-based LCS. The states where even the best actions lead to relatively low rewards have a high risk of not being sufficiently covered by the classifier population since all classifiers matching to such states have low fitness, associated with fewer reproduction opportunities and a higher chance of being deleted. In XCS, classifier fitness is based on the payoff prediction's accuracy. An accurate classifier matching to low-rewarding states can have the same fitness as an accurate classifier matching to high-rewarding states. In essence, strength-based LCSs allocate classifiers primarily to states where a high reward can be obtained, while XCS allocates classifier evenly over the input space.

Generalization. Strength-based LCSs aim at generalizing the classifiers over all states in which they propose the best action, i.e., the action that

	A_1	A_2
S_1	1,000	0
S_2	0	100

Table 3.2: Biased reward function of an environment with two states S_1 and S_2 and two available actions A_1 and A_2.

leads to the highest payoff. In contrast, XCS generalizes its classifiers over all states where the action results in a similar payoff. In this sense, it seems that strength-based LCSs have a more problem-solving-oriented classifier generalization, as the goal of every reinforcement learner is to take the best action in each input state. However, there are no barriers in strength-based LCSs that prevent strong classifiers with high payoff predictions from generalizing into environmental niches for which they do not propose the best-suited action. This problem is termed overgeneralization and is most distinct in environments with a biased reward function.

As a minimal demonstrative example, consider a single-step environment with only two states S_1 and S_2, two actions A_1 and A_2, and the reward function shown in Table 3.2. In state S_1, the action A_1 is correct and leads to a reward of 1,000, while the selection of A_2 is incorrect and yields a reward of 0. In state S_2, action A_2 is correct, resulting in a reward of only 100, while the wrong action A_1 also leads to a reward of 0. In a strength-based LCS, a classifier that matches only to S_1 and proposes the action A_1 will have a fitness equal to its payoff prediction of 1,000. A classifier that matches S_2 and proposes action A_2 will have a fitness of 100. Both classifiers are correct in the sense that they propose the best action for each environmental state, and no further generalization is possible. However, a classifier that is generalized and matches to both S_1 and S_2 and proposes action A_1 will have, on average, a fitness of 500, as it is in 50 % of the cases proposing the correct action in S_1, while it is proposing the wrong action in S_2 in the other cases. Even though it is not proposing the correct action in S_2, it still has a fitness considerably higher than the classifier that proposes the correct action in S_2. Consequently, if S_2 is encountered, the wrongly generalized classifier will dominate the action selection, and action A_1 will be selected. Even worse, the correct classifier matching S_2 has a low chance of being reproduced by the GA and a high chance of getting deleted from the population due to its low fitness. In strength-based LCSs, the best action is unlikely to be consistently selected in low-rewarding niches, as overgeneralized classifiers from high-rewarding niches often dominate the classifiers proposing the best actions.

In XCS, this does not happen, as the magnitude of the received payoff is not influencing classifier fitness. Instead, both classifiers that only match one state and propose the correct actions are assigned a high fitness because the rewards received can be accurately predicted. The overgeneralized classifier has low fitness since the reward it receives is

not consistent and thus cannot be predicted accurately. The degree to which XCS accepts varying rewards can be controlled with its parameter configuration, most prominently by the error threshold ϵ_0. Therefore, XCS can also be faced with overgeneralization, especially if wrongly parameterized, but the problem is far less distinct than in strength-based LCSs. On the other hand, environments with stochastic rewards pose a significant challenge to XCS, as it is impossible to predict the reward that will be received accurately. Whether fitness pressure can still emerge depends on the environment and its amount of generalization opportunities, the stochasticity and shape of the reward function, and the parameter configuration of XCS. Hence, strength-based LCSs are, in general, more robust when faced with stochasticity since the varying payoff predictions of its classifiers have not much effect on classifier fitness as long as "good" and "bad" actions can still be distinguished based on the magnitude of the prediction.

Input/Action Mapping. Strength-based LCSs focus on classifiers with the highest payoff predictions, which is why they tend toward evolving classifier populations that represent best-action maps, i.e., populations that store only the classifiers which propose the best action for each input state. Classifiers that propose non-optimal actions have lower fitness and will be deleted from the population. XCS, on the other hand, evolves populations that represent complete state/action maps. This means that the population can accurately predict the reward for each possible pair of input state and action, as classifiers that propose non-optimal actions are kept in the population as long as their payoff prediction is accurate. The difference between the evolved input/action maps has two significant implications.

First, strength-based LCSs tend to evolve smaller classifier populations. Since mainly the classifiers that propose the best actions are kept in the population, this can lead to a minimally sized population that can still select the best action in each state. In addition, strength-based classifier systems can generalize over all states where the action is optimal, while XCS only generalizes over states if the received reward is sufficiently similar. Thus, strength-based classifier systems can have more generalization opportunities, leading to more generalized classifiers and smaller populations. However, as outlined above, the stronger generalization in strength-based classifier systems can lead to overgeneralization more easily.

The second implication concerns the adaptivity of the LCS. While the additional classifiers with non-optimal actions that XCS stores may not add to the optimal solution itself, they can have beneficial effects regarding adaptivity and exploration control. The complete state/action mapping of XCS ensures that even for non-optimal actions proper generalizations have been evolved, which can improve the speed with which a classifier system adapts to environmental changes. In [40], XCS has been compared to a strength-based LCS on a binary classification problem that, after some time, underwent an abrupt change that flipped all classes.

XCS adapted quickly, while the strength-based LCS took considerably longer. Since the change in the environment did not affect the way generalizations can be applied in the environment, XCS made use of its complete input/action mapping and only updated the payoff predictions of existing classifiers. The population of the strength-based classifier system did not have classifiers available that proposed the previously wrong actions. Due to this, it had to create entirely new classifiers via its GA. However, the evolutionary search for new classifiers that are properly generalized takes considerably longer than merely updating the parameters of existing classifiers.

The control of exploration is linked to the Explore/Exploit (E/E) dilemma in reinforcement learning, which is investigated in the context of XCS more detailed in Chapter 4. For a reinforcement learner to learn the best actions via interaction with the environment, it must explore all available actions to determine which one is the best suited in a given state. Trying out actions other than the apparently best will likely harm operational performance in the short term. However, it might generate new knowledge, which can improve the overall solution in the long term. Hence, a reinforcement learner must perform both exploration and exploitation, which is known as the E/E dilemma. Typically, a desirable behavior is that the learner performs exploration at the beginning of learning and transitions to exploitation once the optimal solution is found. As argued by Kovacs [51], strength-based LCS are less suited for such exploration control because their classifier populations focus only on classifiers with the (apparently) best actions. However, it can never be determined with certainty whether the current classifiers propose the best actions or if there might exist even better actions for some states, which are not included in the population because they have not yet been explored. Hence, a strength-based classifier system cannot determine when exploration should be stopped in favor of exploitation – at least not based solely on the status of its classifier population. In XCS, however, a complete input/action mapping is evolved. Once the whole input space is covered for all actions with classifiers having accurate payoff predictions, it can be determined with certainty that the optimal solution is found and exploration can be stopped.[2]

Multi-step Environments. Due to the reward discounting mechanism, multi-step environments have an inherently biased payoff function. As outlined above, this will often lead to overgeneralization in strength-based LCSs, while XCS copes better. In addition, the complete input/action mapping of XCS is also favorable for multi-step environments. Even though the optimal strategy requires always selecting the action that leads to the largest payoff, which can also be achieved with a strength-based LCS, learning the optimal strategy will frequently require selecting apparently non-optimal actions to discover the best path toward the goal state. In strength-based LCSs, classifiers proposing (apparently) non-optimal ac-

2 An exception are dynamic environments, which are discussed in Chapter 4 in the context of the E/E dilemma.

tions are not preserved in the population for a long time, which prevents finding the best strategy in the long term. Overall, strength-based LCSs are considered to be incapable of solving even simple multi-step problems. XCS is coping considerably better with multi-step environments, even though it can also be affected by effects such as overgeneralization [58].

3.3 XCS EXTENSIONS

One popular point of view on LCSs is to see each classifier system as a combination of different building blocks such as condition structure, GA, and classifier parameters [94]. With the proper selection of building blocks, LCSs can be tailored to various problem environments. Since this not only applies to LCSs in general but also to XCS in specific, the flexibility of XCS is one of its main postulated strengths. Throughout the past decades, many modifications and extensions have been proposed to the original version of XCS as presented in Section 3.1. This section gives a non-exhaustive overview of XCS research, focusing on extensions that enable XCS to solve additional types of problem environments or improve its behavior when faced with challenges typically associated with environments that benefit from self-* properties.

Condition Structure. One significant restriction of XCS is its binary condition structure. When faced with an environment emitting real-valued inputs, the inputs must be discretized, e.g., with a binning approach, to be converted into a bitstring. A high resolution leads to a long bitstring, which increases the required population size and the number of samples XCS needs until the optimal population is evolved. On the other hand, a discretization into a few bins risks that the input is discretized too coarse-grained to evolve a sufficiently good solution. Hence, several attempts have been made to enable XCS to cope natively with real-valued inputs. The resulting classifier system is commonly referred to as XCSR. The most widespread approach is to use conditions with an interval defined for each real-valued number of the input [97, 113]. Only if all numbers of the input lie in the intervals defined by the condition does the classifier match. The ordered bound representation, as discussed in [97], is the most popular representation for real-valued inputs in XCS and is also used in the case study presented in Chapter 6. For each real-valued number x_i of the N-dimensional input, the classifier condition has an interval predicate $[l_i, u_i]$, where l_i is the lower bound and u_i is the upper bound. If all input numbers x_i lie inside the respective intervals, i.e., $l_i \leq x_i \leq u_i$, the classifier is said to match. This way, each classifier covers a hyper-rectangular subspace of the N-dimensional input space. The system designer must specify the minimal and maximal values that can occur at the input. Typically, all inputs are normalized to the range between 0 and 1. Upon creating a new classifier via covering, the intervals are randomly determined around the current inputs with $l_i = x_i - U[0, s_0]$ and $u_i = x_i + U[0, s_0]$, where U is a uniform probability distribution that chooses a value between 0 and s_0, which is a configurable hyperpa-

rameter. Crossover works the same as with binary conditions and can occur within and between an interval predicate. If mutation is applied, a random amount $\pm U[0, m_0]$ is added to one of the interval bounds, where m_0 is another configurable hyperparameter.

To cover non-linear subspaces, ellipsoidal conditions have been employed [19], with promising results in environments where the niche boundaries are not linear. To discern complex non-linear subspaces, neural networks have been proposed to determine if a classifier matches [15]. In such a Neuro-XCS, each classifier has a feed-forward neural network instead of a condition. The input is fed into the neural network, and its output determines whether the classifier matches. While this allows classifiers to cover nearly arbitrarily complex subspaces, it also increases the learning effort required to evolve the population, as the GA is also evolving the weights of the neural networks. In addition, the rules evolved by Neuro-XCS are no longer human-interpretable.

Stochastic Environments. Since classifier fitness in XCS is based on the accuracy of the payoff prediction, environments with non-deterministic rewards can pose a challenge. A stochastic reward cannot be predicted accurately, which inevitably negatively influences classifier fitness. Even though there is experimental evidence that XCS can cope with small variations of the reward [60], larger uncertainty prevents XCS from evolving any reasonable solution. With large, unpredictable reward variations, the prediction accuracy drops considerably. However, XCS is not able to distinguish between inaccuracy that is caused by stochasticity and inaccuracy that is caused by wrongly generalized classifiers. Essentially, fitness pressure in XCS is no longer pushing towards accurate and maximally generalized classifiers but becomes undirected. Lanzi and Colombetti [60] have been the first to investigate this challenge and proposed an approach that separates the observed prediction error into ϵ_{env}, which is the error induced by the stochastic environment, and ϵ_{gen}, which is the error induced by wrong generalizations. They introduce a new parameter μ, which serves as an estimate for the prediction error caused by the stochasticity of the environment and is determined based on the minimum prediction error of all classifiers in the current match set.[3] When updating the prediction error ϵ, μ is subtracted from the observed prediction error, such that the remainder should mainly consist of the prediction error caused by incorrect generalizations.

More recently, Tatsumi et al. [100–103] have proposed a series of modifications to deal with different kinds of stochasticity in the problem environment. In [100], each classifier has a variable error threshold ϵ_0 that is adjusted based on the standard deviation of the received rewards. This way, the inherent variations of the reward are accounted for when determining classifier accuracy. The standard deviation of the reward is estimated for every single state/action pair individually, and only after the standard deviation meets a convergence criterion, ϵ_0 is updated. To avoid recording the standard deviation for all state/action pairs, [102]

3 Not to be confused with the hyperparameter μ, which denotes the mutation probability.

uses the standard deviation recorded for each classifier as the basis for its fitness. Additionally, a subsumption mechanism is introduced that determines accuracy based on the range of rewards a classifier received in the past. To tackle stochasticity in environments with discrete reward levels, XCS-CR [101, 103] uses classifiers that count how often each possible reward has been received. Together with a variable value of ϵ_0 that is unique to each classifier, the accuracy of a classifier is determined based on the difference between its average reward received and the reward value that has been received most frequently.

However, all presented approaches have so far been evaluated only on simple toy problems with idealized assumptions. Moreover, many approaches have relatively restrictive requirements, e.g., a discrete reward function [101] or a complete sampling of the state/action space [100], while others negatively impact the learning mechanism in XCS, e.g., by fostering overgeneralization [60, 101]. Hence, a more widely applicable mechanism to enable XCS to cope with stochastic environments remains an open topic for future work.

Non-Markovian Environments. Multi-step environments are categorized into being either Markovian or non-Markovian. In Markovian environments, the current state can be determined solely based on the current sensory input. A non-Markovian environment, on the other hand, has at least two aliased states, which lead to the same input to the learner even though they are different states. The environment is said to be only partially observable, and the learner is affected by perceptual aliasing, as its perceived input alone is not sufficient to determine the current state and the appropriate action reliably. Since XCS uses only the current sensory input to determine the action to be executed, it can solve Markovian environments. However, it will generally fail to evolve an optimal solution for non-Markovian environments. All modifications that attempt to tackle this restriction include adding memory to XCS, such that it can "remember" from which state(s) it has entered the current state. This awareness of the past then enables XCS to distinguish between aliased states. The first approach that has been proposed consists of adding an internal register with a configurable number of bits to XCS and extending each classifier with an internal condition and an internal action [55, 57, 59]. The internal action is used to modify the internal register. XCS can then evolve classifiers that, when entering aliased states, set the internal register to different values depending on which aliased state is entered. The classifiers matching to aliased states then use the internal condition to discern between the different aliased states. A more straightforward way to tackle the problem of perceptual aliasing is to extend the input of XCS with the sensory input perceived in previous states [81]. This way, information about the past is explicitly added to the input of XCS, and previously aliased states are now distinguishable based on the current input. However, this increases the size of the condition for all classifiers in the population, even if they match only to non-aliased states. To cir-

cumvent this drawback, [115] proposes an approach where only a subset of all classifiers has memory-extended conditions.

Naturally, all outlined approaches have specific (dis-)advantages. For instance, XCS with an internal register can carry the memory over an unrestricted number of steps, but the learning process can be unreliable, as it requires two classifiers to cooperate in distinguishing aliased states, i.e., one classifier that sets the internal register when entering an aliased state and one classifier that matches to the aliased state and the appropriate value of the internal register. On the other hand, extending the classifier's condition with previous inputs is a straightforward approach that does not require extensive modifications of XCS but dramatically increases the input space, which often requires a larger population and more samples to evolve the optimal solution. Overall, all outlined approaches can solve simple non-Markovian environments with few aliased states but quickly reach their boundaries in more complex environments. For such cases, Anticipatory Learning Classifier Systems (ACSs) [18] are typically employed, which are another class of LCSs and not direct descendants of XCS. In ACSs, each classifier anticipates the next state, allowing for look-ahead path planning in more complex non-Markovian environments.

Function Approximation. Besides being used as an RL agent in environments requiring action selection or classification, XCS can also approximate functions. XCSF [114] uses classifiers with the real-valued conditions of XCSR but has only one dummy action available. Instead of the action, the payoff predicted for the current input is outputted as an approximation of the function. Since each classifier has a payoff prediction that is invariable over the input space it matches, the resulting function approximation of the whole classifier population can be relatively inaccurate. To overcome this, the prediction of a classifier is computed based on the current input. Each classifier is extended with an (N+1)-dimensional weight vector w that is multiplied with the N-dimensional input augmented by a constant. That way, each classifier forms a linear approximation of the input subspace it covers, and the resulting approximation of the whole classifier population is piecewise-linear. The weight vector w is learned at runtime with a gradient technique, i.e., a modified delta-rule. Extensions to XCSF have been focused mainly on achieving non-linear approximations, e.g., with polynomial approximations [62] or feed-forward neural networks [61], and non-linear condition spaces, e.g., with ellipsoidal conditions [21]. If more than a single dummy action is used, XCSF can also be used for its original purpose to learn an optimal strategy in an RL environment. It can outperform XCSR, as the computed predictions that vary over the input space can be more accurate than the constant predictions of XCSR. However, the weight vectors used to compute the prediction must be learned in addition to evolving classifiers with properly generalized conditions. This poses an additional burden that can increase the time until the optimal solution is learned.

Self-aware Computing Systems. As XCS is frequently proposed to be employed in autonomous and self-adaptive systems, an existing body of research has proposed extensions to improve the applicability of XCS in such application scenarios, e.g., in the context of Organic Computing (OC). Since the primary goal of self-adaptivity is to adjust to unforeseen or changing environmental conditions, most works focus on improving the online learning behavior of XCS. One widespread characteristic of real-world problem environments is a non-uniform sampling of the input space, i.e., some states frequently occur while others are observed rarely. However, XCS is expected to perform well even in unknown environmental conditions that rarely occur. To achieve this, Stein et al. proposed the concept of classifier interpolation [95]. When faced with a situation where not enough matching classifiers are present in the population, e.g., because the observed state has never been encountered before, classifiers are not generated randomly based on the standard covering procedure. Instead, existing classifiers with conditions similar to the current input are used to determine the parameters of the newly generated classifier. Assuming that environmental niches with similar sensory inputs share enough commonalities, XCS can perform adequately even when faced with unknown situations. With a similar goal, an experience replay mechanism [93] has been proposed that stores past inputs and the received reward to feed it into XCS again at a later time. This way, classifiers matching to an infrequently observed state can be further optimized before the state is encountered again.

The concept of active learning is introduced into XCS in [92]. With active learning, XCS is made "curios" and can query an external oracle for guidance in environmental niches where it has insufficient experience. Thus, XCS can learn appropriate behavior even before the situation is encountered. The oracle can be, for instance, a heuristic, an environmental simulation, or a human-in-the-loop. The use of domain knowledge to bootstrap the classifier population is considered by Feist et al. [30], who propose a rule language that system designers can use to specify a ruleset and that is more intuitive than manually specifying classifiers. After the specification, the ruleset is automatically converted into rules compatible with an LCS.

3.4 XCS FOR COMPUTATIONAL SELF-AWARENESS

The concept of computational self-awareness encompasses a high degree of autonomy and adaptability to adjust to unknown circumstances without external help. Such properties are commonly associated with "systems" from nature, most distinctively with animals. In concepts such as Organic Computing (OC), the inspiration from nature to create life-like adaptive systems is even explicitly considered in the terminology. XCS combines Evolutionary Computation (EC) and Reinforcement Learning (RL), where EC is inspired by Darwinian evolution, while RL describes the concept of learning through interaction with the environment,

which is employed by nearly all living beings that are considered to be at least remotely intelligent. Since both learning approaches mimic how learning and optimization occur in nature, XCS seems to be well suited for implementing self-awareness – even at first glance and without more profound knowledge of the algorithmic details of XCS, its strengths, or weaknesses.

The RL mechanism of XCS is based on Q-learning, a tabular RL technique. All state/action pairs are enumerated in Q-learning, and the so-called Q-value is estimated for each pair. In single-step problems, the immediate reward is estimated, while in multi-step problems, the payoff is estimated, i.e., the sum of the immediate reward and discounted future reward. Since XCS uses classifiers covering whole input subspaces, it is essentially a generalizing Q-learner. While evolving proper generalizations through the GA can be a time-consuming process, XCS also lifts some disadvantages of Q-learning, as it can alleviate the Q-table's exponential growth in the input size. Further, XCS can identify environmental patterns and learn the optimal policy for the environment even if it has not yet seen all possible states. This capability is an important property for self-aware systems, as XCS can, in principle, propose optimal, or at least suitable, actions even in so far unknown situations, which is unachievable with Q-learning. Naturally, there is no guarantee that XCS will act appropriately in unknown situations, but this is true for all learning algorithms and is achievable only through the use of domain knowledge.

In contrast to many other machine learning techniques, the ruleset evolved by XCS is, to a high degree, human-interpretable, which gives XCS a high level of explainability by design. Even though it cannot be explained why the stochastic evolutionary process of the GA has evolved the rules the way they are, it is still possible to analyze the population for patterns, identify knowledge gaps, or manually inject domain knowledge via hand-crafted classifiers. This can be especially useful for self-aware Cyber-Physical Systems (CPSs) [9], which do not operate in virtual environments but have a possibly harmful impact in the real world. Leveraging the interpretability of XCS's rules, Chapter 5 proposes the concept of forbidden classifiers, which are a special type of hand-crafted classifiers that prevent the selection of actions that are considered to be a safety violation in the current situation. If specified appropriately, they guarantee that XCS is not violating any safety requirements even in the face of so far unseen situations – regardless of whether the action selection of XCS has been based on exploration or exploitation.

As described in Section 3.2, XCS is said to be well-suited to handle exploration, which is required for highly autonomous RL agents. The complete state/action map that is evolved and the corresponding prediction errors give an overview of how well XCS has learned the characteristics of the environment. Since each classifier forms a local solution that matches only to a subspace of the input space, it is possible to determine how well XCS has covered different environmental niches and where further exploration is required. This enables XCS to emit a

highly adaptive learning behavior. If the environment is non-stationary and only a part of it is affected by a change, a scenario investigated in Chapter 4, exploration can be applied in the affected niches only. The GA in XCS is a steady-state niche GA, which means that it acts continuously as long as exploration is enabled and evolves classifiers that match to the environmental niches that changed. This leaves classifiers of unchanged niches mostly unaffected, which preserves the operational performance in these regions of the environment.

The extensions and modifications of XCS outlined in Section 3.3 enable XCS to tackle a wide variety of different problem environments. This flexibility is not only observed for XCS but also for other LCSs, which has led to the assertion that the learning technique LCS is "a jack of all trades, but master of none" [106]. Indeed there has not yet emerged a distinct "killer application" where XCS, or any other LCS, achieves results distinctly superior over all other learning approaches, except for toy problems such as the 135-Multiplexer [107]. However, this does not pose to be a dealbreaker for the use in self-aware computing systems. Finding the optimal learning algorithm for a given task often requires extensive (empirical) analysis, potentially with lots of expert domain knowledge or large datasets. In the application domains envisioned for self-aware systems, this is typically infeasible. Hence, a learning technique that applies to a wide range of application environments and achieves good but not necessarily optimal results will generally be preferred over a diverse set of highly specialized techniques.

Overall, XCS seems like a natural fit for deployment in self-aware systems due to its (1) adaptivity, (2) interpretability, and (3) flexibility which allows its deployment in a wide range of applications. This is also why it is already considered frequently in related domains such as Organic Computing (OC). This includes both the incorporation of XCS and other LCSs into the design frameworks [73, 104], and the use in applications, e.g., for traffic control [83], test case prioritization [87], or smart camera management [96].

4

EXPERIMENTAL COMPARISON OF AUTONOMOUS EXPLORE/EXPLOIT STRATEGIES

Computational self-awareness and related design paradigms aim to move design-time decisions to the runtime and into the system's responsibility, enabling a technical system to cope with changing environments and unforeseen situations on its own. To achieve this, a system needs to be both autonomous and adaptive, as it must be able to make decisions on its own (*autonomy*) and also adapt its decision-making to changes in the operational environment (*adaptivity*). However, when XCS is employed as an adaptive decision-making engine, it is affected by the Explore/Exploit (E/E) dilemma, like every other reinforcement learner [99]. During operation, a reinforcement learner is constantly faced with deciding whether to exploit the existing knowledge by taking the most promising action or to deliberately select an action that is not the apparently best to potentially gain additional knowledge. Hence, the E/E decision is concerned with the action selection and resides in the self-expression engine of the reference architecture for computational self-awareness.

The E/E decision constitutes a dilemma since obtaining new knowledge through exploration incurs a short-term performance loss, while too much exploitation of the already learned knowledge risks staying on an unnecessarily low level of performance in the long term. In static application domains, a fixed schedule that, over time, decreases exploration in favor of exploitation might be sufficient. However, in the face of environmental dynamics, previously obtained knowledge can become obsolete, requiring frequent shifts from exploitation back to exploration to preserve adaptivity. To reach full autonomy and adaptivity, autonomous E/E strategies are required to let reinforcement learners decide on their own what to pursue, depending on the current state of the environment and their learning progress

For XCS, its inventor Wilson formulated a high-level overview of ten different E/E strategies [111] about 25 years ago. The most crucial distinction between the strategies is whether they are global, i.e., based on global metrics of the whole classifier population, or local, i.e., determine the E/E decision for each situational input individually. Even though the importance of reliable E/E strategies for XCS was identified more than two decades ago, not much research has been conducted in this direction since then. Instead, an ϵ-greedy strategy with a fixed exploration

probability of 0.5 is commonly used, which is entirely sufficient for conducting research on the learning mechanisms of XCS. The frequent exploitation cycles allow for monitoring the development of its learning progress online, and comparability to other works is preserved. For implementing autonomous agents, however, such a static approach is infeasible. Still, the ϵ-greedy is frequently employed even in research on autonomous systems, e.g., in the domain of Organic Computing (OC) [96], by neglecting exploration cycles during the performance evaluation – an approach clearly infeasible for the practical deployment of self-aware systems.

Among the few autonomous E/E strategies that have been developed, the majority have mostly been evaluated in a single scenario only and without comparison to other strategies. Further, their parameterization is often discussed on a qualitative level only, giving practitioners no guidelines on how to apply the strategies to other learning problems. This experimental study aims to narrow the gap in research on autonomous E/E strategies for XCS by experimentally comparing one local and three global E/E strategies proposed in the literature. The evaluation takes place on learning problems well known in the Learning Classifier System (LCS) community, i.e., multiplexer and maze environments. An automated parameter optimization for identifying suitable parameter configurations is a substantial part of this evaluation. The optimization takes place not only on each environment separately but also over all environments simultaneously to identify possible "one-fits-all" configurations. Further, we evaluate the behavior of the strategies in the presence of environmental dynamics. This chapter is, in slightly modified form, currently under review by the Soft Computing journal [39] and represents a substantially enhanced and extended version of our paper published at the International Workshop on Learning Classifier Systems (IWLCS 2021) [37]. In summary, this chapter makes the following contributions to the existing body of research:

- We present the first experimental comparison of E/E strategies for XCS. The results have been gathered in different problem environments and show that the strategies depict vastly different E/E behaviors, both in static and dynamic environments.
- To find suitable hyperparameters for each E/E strategy, an automated parameter optimization has been conducted. We find that no strategy has a "one-fits-all" configuration that is optimal for each evaluated scenario. However, even though our study employed only artificial problems common in LCS research, we suppose that XCS practitioners planning to use an E/E strategy can still benefit from the found hyperparameter sets by relating the characteristics of the application domain to the scenarios evaluated in our study.
- The experimental evaluation shows that the local error-based strategy seems most suitable for use in self-aware systems, as it depicts

a robust E/E behavior, reacts adequately to environmental changes, and is the easiest to parameterize. However, it is unsuited for multi-step environments with sparse rewards.

This chapter continues in Section 4.1 by describing the methodology of the literature study and the selected E/E strategies. Our experimental setup is outlined in Section 4.2, while Section 4.3 discusses the experimental results gathered with parameters optimized for each environment separately. The sensitivity to non-optimal parameters is discussed in Section 4.4, where an attempt toward a "one-fits-all" parameterization has been made and evaluated. In Section 4.5, the performance of each strategy is investigated in the face of environmental dynamics. Finally, Section 4.6 summarizes the main findings and guidelines for successfully employing an E/E strategy, while Section 4.7 concludes the paper and outlines future work.

4.1 EXPLORE/EXPLOIT STRATEGIES

The study focuses on E/E strategies that enable autonomy and determine *when* exploration is called for and not *how* the exploration should take place. Consequently, we focus on schemes that decide between employing pure exploitation, i.e., taking the action with the highest payoff prediction, or pure exploration, i.e., choosing a random action. Directed or biased exploration, e.g., by selecting actions that promise the largest gain in knowledge, is not part of this study, and an investigation of directed exploration is left for future work. Nevertheless, with most directed exploration techniques, the question of when exploration is called for must also be answered. In addition, we restrict the study to E/E strategies specifically designed for XCS, as we deem strategies tailored to its unique learning mechanism as most useful.

4.1.1 *Literature Study*

The primary tool for searching published works on E/E strategies for XCS was Google scholar, and the results represent the state as of 7th September 2022. We have employed three different search terms, namely *XCS "exploration exploitation"* (141 result entries), *XCS "exploration strategy"* (68 entries) and *XCS "explore exploit"* (169 entries). The terms in quotation marks must match exactly but can still encompass special characters, e.g., "explore exploit" matches the phrase "explore/exploit". Further, we have considered all publications that, according to Google scholar, cite the seminal paper of Wilson [111]. Overall, this resulted in two E/E strategies that match our criteria, i.e., the HECS strategy of McMahon et al. [71] and the meta-rules strategy of Rejeb et al. [85]. Further, we have searched all publications that reference these publications or are cited by them, but with no additional results. In addition to the HECS and Meta-rules strategies, we have selected one global and one local error-based strategy

from Wilson [111], which have also served as a baseline for comparison in [85].

Not considered due to the selection criteria has been the work of Bagnall and Smith [8], who propose a strategy that not only determines when to explore but combines it with directed exploration through temperature-based Boltzman weighting. Also not considered has been the E/E strategy based on a fuzzy system proposed by Hamzeh and Rahmani [34] since it requires knowing the system's lifetime, represented by the number of total XCS iterations. For truly autonomous systems, we assume this information to be unknown or at least associated with a high degree of uncertainty. The strategy used by Liu et al. [67] continuously decreases exploration without any feedback related to the current state of learning, thereby making it unsuitable for implementing autonomy.

4.1.2 *Selected E/E Strategies*

Meta-rules [85]. Exploration and exploitation are balanced by executing repeated cycles of n exploration runs, followed by m exploitation runs. After each such cycle, the performance during the exploration and exploitation runs, denoted by $Perf_{explore}$ and $Perf_{exploit}$, respectively, is used to increase or decrease the number of exploitation runs m. This is done by the following two meta-rules, where e_r is termed the exploration rate:

$$m = \begin{cases} m \cdot (1 - e_r) & \text{if } Perf_{explore} > Perf_{exploit} \\ m \cdot (1 + e_r) & \text{if } Perf_{explore} \leq Perf_{exploit} \end{cases} \tag{4.1}$$

The meta-rules balance exploration and exploitation by adapting the ratio of exploration to exploitation runs while maintaining a minimum amount of exploration by keeping n constant. To ensure that the intervals between exploration periods are not growing too large, a maximum value of m can be specified. In multi-step environments, n and m represent the numbers of complete runs and are unaffected by the number of steps done in each run. Overall, the strategy is parameterized by three values: The number of exploration runs n, the initial value of exploitation runs m, and the exploration rate e_r, ranging from 0 to 1, which controls the change of m.

Global Error [111]. At the beginning of a run, it is decided if an exploration run is conducted according to the exploration probability $p_{explore}$. As opposed to the common ϵ-greedy strategy, the exploration probability is not fixed but determined as

$$p_{explore} = min(1, G \cdot E_{global}) \tag{4.2}$$

where E_{global} is the moving average of the global prediction error and G a configurable gain factor. The global prediction error is the absolute difference between the predicted and received payoff. In our implementation, we use the values in the prediction array as the predicted payoff.

The strategy is parameterized by two parameters, namely the gain factor G and the moving average's windows size W. To make gain factors between environments with different reward schemes comparable, E_{global} is normalized to the range of payoffs. In multi-step environments, the prediction errors during a run, based on the internal payoffs and not the immediate rewards, are averaged and inserted into the window as a single value to avoid that runs have a different impact on the global prediction error just because they are shorter or longer. If a multi-step environment does not consist of distinguishable runs after which the problem is solved but instead continues endlessly, this scheme needs to be modified, e.g., by inserting the prediction error of each step.

Local Error [111]. Since it is a local strategy, the exploration probability $p_{explore}$ is determined individually for each received input. For each action in the match set, the numerosity-weighted average of the prediction errors ϵ is calculated. The average over the actions' error estimates is then used to set the exploration probability as

$$p_{explore} = min(1, G \cdot E_{local}) \tag{4.3}$$

where E_{local} is the average over the actions' error estimates, and G the gain factor, which is the only configurable parameter of this strategy. Again, E_{local} is normalized to the range of payoffs.

HECS [71]. The HECS strategy was developed specifically for multi-step problems and distinguishes between two different exploration levels. The *accuracy-induced* exploration level E_A ranges between -1 and +1, with an initialization of 0, and is updated every step with

$$\Delta E_A = \Delta E_{max} \cdot F \cdot \left(\frac{2}{1 - e^{M_{PA}}} \cdot \left(e^{|O_s| \cdot M_{PA}} - 1 \right) + 1 \right) \tag{4.4}$$

where ΔE_{max} is the maximum possible change, F the fitness of the prediction, O_s the payoff over-/undershoot (positive/negative prediction error) scaled to the maximum range of payoffs, and M_{PA} a parameter that defines the tolerance to perfect accuracy. The exploration probability that is used to decide if an exploration run is conducted is determined as

$$p_{explore} = \left(0.5 - \frac{L_{exploit} \cdot E_A}{2} \right) \tag{4.5}$$

where $L_{exploit}$ takes a value between 0 and 1 and, for values below one, assures a minimum level of exploration. As fitness of the prediction, we take the numerosity-weighted average of the classifiers' fitnesses in the action set.

In multi-step problems, a second, *reward-induced* explorer level E_R is used to escape from unsuccessful exploit trials. First, the maximum number of steps required to reach a reward is estimated as $n_{max} = log_\gamma \left(\frac{P_{min}}{P_{max}} \right)$, where P_{min} and P_{max} are the minimum and maximum predictions present in the population and γ the discount factor of XCS. In case P_{min} is negative or zero, n_{max} must be sanitized to a meaningful value. After n_{max}

steps have passed, the reward-induced explorer level E_R, initialized as 1, begins to change according to

$$\Delta E_R = \Delta E_{max} \cdot \frac{1}{e^{M_{RS}} - 1} \cdot sign(R_S) \cdot \left(e^{|R_S| \cdot M_{RS}} - 1 \right) \tag{4.6}$$

where R_S is the immediate reward that is received scaled to the maximum reward and M_{RS} is a scaling factor that affects the sensitivity of ΔE_R to the magnitude of R_S. Hence, a positive reward increases E_R, while negative rewards (punishments) decrease E_R. During an exploitation run, HECS switches to exploration mode with a probability $P_{SwitchToExplr} = 1 - E_R$. Hence, a series of exploitation steps without positive reward increases the probability of switching to exploration. Overall, the HECS strategy is parameterized by four values: ΔE_{max}, $L_{exploit}$, M_{PA} and M_{RS}.

4.2 EXPERIMENTAL SETUP

The experimental comparison of the E/E strategies takes place in two static scenarios, i.e., as evaluation and optimization in each environment separately and over multiple environments. Further, we investigate a dynamic scenario in which the environments change during the experiment. Common to all scenarios are the benchmark environments described in Subsection 4.2.1 and the parameter optimization procedure outlined in Subsection 4.2.2. The methodology of gathering the experimental results for comparing the strategies is presented in Subsection 4.2.3.

4.2.1 *Benchmark Environments*

11-Multiplexer. The 11-Multiplexer is a single-step problem in which XCS receives 11 input bits, of which the first three are index bits that point to one of the remaining eight bits that XCS has to predict. If XCS outputs the correct bit value, it receives a reward of 1,000 and, otherwise, a reward of 0. With the employed 30,000 iterations, it represents a rather simple problem, as XCS with the common ϵ-greedy strategy and a fixed exploration probability of 0.5 can fully solve the problem in its exploit trials after roughly 10,000 iterations. Hence, a well-suited E/E strategy is expected to do some exploration at the beginning and then switch to full exploitation once XCS can solve the problem completely.

20-Multiplexer. The 20-Multiplexer is similar to the 11-Multiplexer problem but more complex to solve due to the larger input space, as XCS receives an input of 20 bits containing four index bits. With the same number of 30,000 iterations, XCS with the common ϵ-greedy strategy is not able to derive a complete solution. Instead, it achieves a classification accuracy of roughly 85% at the end. Hence, the 20-Multiplexer represents a case in which XCS cannot derive a complete solution and tests the ability of the E/E strategies to carefully balance exploration and exploitation during the whole time to maximize overall performance.

```
R R R R R R R R
R . . R . . (F) R
R R . . R . . R
R R . R . . R R
R . . . . . . R
R R . R . . . R
R . . . . R . R
R R R R R R R R
```

Figure 4.1: The Maze4 environment. Empty fields are denoted by dots, while obstacles are represented by rocks ('R'). The target field is the food ('F') in the upper right corner.

```
R R R R R R R R R R R R R
R R . . . R R R . R R . R
R . R R R . R R . R . R . R
R . R R R . R . R R R . R R
R (F) R R R . R R . R R R R R
R R R R R R . . R R R R R
R R R R R R R R R R R R R
```

Figure 4.2: The Woods14 environment. The target field is the food ('F') in the lower left corner.

Even though XCS can solve large multiplexer problems, such as the 135-Multiplexer [75], we have stuck with the smaller variants, as we expect that all multiplexer problems depict a similar E/E behavior. For solving large multiplexers, XCS requires specifically tailored configurations and substantially longer execution times, which would have impeded the automatic parameter optimization of the E/E strategies.

Maze4. Maze environments are multi-step problems in which XCS navigates a robot through a maze to reach a target, denoted as food. Shown in Figure 4.1 is the Maze4 environment. As input, XCS receives the types of the eight surrounding fields (empty, rock, or food) and then has to make a step in one of the eight directions. If a step is made towards a rock, the position of the robot does not change. At the beginning of each run, the robot is placed randomly on an empty field, and the goal of XCS is to learn the shortest path to the food from each field on the map. Upon reaching the food, XCS receives a reward of 1,000, and for every other step a reward of zero. In Maze4, the average length of the shortest path is 3.5 steps. It is calculated by determining the shortest path length for each empty field in Maze4 and then averaging over all fields, as each is chosen as starting position with equal probability. XCS finds the shortest path with the ϵ-greedy strategy after roughly 500 runs, with a run ending after the food is reached or 30 steps have passed. However, we have noticed that some E/E strategies require considerably longer to reach a viable solution, so we have employed 3,000 runs.

Woods14. Another maze environment is the Woods14 environment shown in Figure 4.2. In contrast to Maze4, it offers only a single path to the target field. However, this path is considerably longer, as the shortest path consists of up to 18 steps if the robot starts on the empty field in the upper right corner. Therefore, the shortest path has an average length of 9.5 steps. With the ϵ-greedy strategy, this is reached after roughly 2,000 runs, with each run being prematurely stopped after 100 steps if the food has not yet been reached. In our experiments, we have extended the length to 4,000 runs to account for different characteristics of the E/E strategies.

Overall, the four environments evaluate different aspects of the E/E strategies. The 11-Multiplexer environment investigates the capability of a strategy to determine when the classifier population fully solves the problem, and exploration should be stopped in favor of performance-maximizing exploitation. On the other hand, the 20-Multiplexer environment tests how well a strategy can balance exploration and exploitation if the problem cannot be fully solved and neither full exploration nor full exploitation is called for at any time. The Maze4 environment as a multi-step environment should reveal any general behavioral differences in the strategies for multi-step problems. The Woods14 environment represents a multi-step corner case, requiring XCS to learn a long chain of actions to receive a positive reward.

We have used the Python implementation scikit-XCS [118] for all our experiments. We have kept the default parameter settings of scikit-XCS[1] and used a discount factor γ of 0.71 for Maze4 and 0.9 for Woods14. For the 11-Multiplexer and Maze4 problems, a maximum population size N of 800 has been used, while a larger population of $2,000$ classifiers has been employed for the more complex 20-Multiplexer problem. The Woods14 environment made use of a population size of $1,500$ classifiers. Since the multi-step environments, i.e., Maze4 and Woods14, are susceptible to the problem of overgeneralization, we have employed the specify operator [56]: If the classifiers in the action set $[A]$ have been updated at least $n_{sp} = 20$ times and have an average prediction error that is at least twice the average error of the population, a classifier is selected from $[A]$. Each don't care of its condition is specified to match the current input with a probability of $p_{sp} = 0.5$ for Maze4 and 0.8 for Woods14. As an alternative for tackling overgeneralization, the gradient descent technique for parameter updates [19] could have been used. However, we do not expect substantial differences in both approaches' E/E behavior.

All benchmark problems have been implemented as randomized Reinforcement Learning (RL) environments, i.e., the problem instances are randomly generated and not drawn from a data set. It is important to note that we always consider and report the overall performance of

1 That is $\beta = 0.2$, $\alpha = 0.1$, $\nu = 5$, $\mu = 0.04$, $\delta = 0.1$, $p_I = 10$, $\epsilon_I = 0$, $f_I = 0.01$, $P_\# = 0.5$, $\epsilon_0 = 10$, $\chi = 0.8$, $\theta_{GA} = 25$, $\theta_{sub} = 20$, $\theta_{del} = 20$, $\gamma = 0.71$, $DoGaSubsumption = True$, $DoActionSetSubsumption = False$

XCS, i.e., both explore and exploit runs, as opposed to most works in the field that only report performance during exploit runs.

4.2.2 *Parameter Study*

Since the selected E/E strategies do not have any obvious well-suited or even optimal parameter values, we have conducted an automated parameter optimization of the strategies with the tool irace [69]. Table 4.1 summarizes the applied range of allowed parameter values. The parameter m of the meta-rules strategy only represents its initial value since it is adapted at run-time by the strategy. The maximum possible value for m has been excluded from the parameter study and instead has been set to 10% of the total number of iterations. The parameter M_{RS} of the HECS strategy has not been considered in this parameter study, either, as it only applies to multi-step environments. However, in the maze environments, only two possible rewards exist, where the reward of 0 can be considered as punishment. Hence, the value of R_S in Equation 4.6 is either -1 or +1, and M_{RS} does not affect the change of the explorer level. Further, we have set n_{max} to 15 steps for Maze4 and to 50 steps for Woods14, i.e., 50% of the maximum number of steps, in case P_{min} is zero.

As problem instances, 15 different seeds have been used to initialize the random number generators inside the benchmark environments. An important aspect is defining the target metric that irace is optimizing, as no obvious choice exists for assessing the quality of the E/E strategies. In general, it depends on the application of how the performance of XCS is quantified. For instance, in some environments, the performance at the beginning can be close to irrelevant, e.g., if the system has an initial setup period. However, in other cases, the performance is equally important over the whole runtime. We have opted for a tradeoff between these two cases by taking all iterations into account for calculating the optimization metric but assigning earlier iterations a smaller weight, as shown in Equation 4.7. The performance during the first quarter of all iterations N is assigned a weight of $\frac{1}{15}$, the second quarter a weight of $\frac{2}{15}$, the third quarter a weight of $\frac{4}{15}$ and the last quarter the highest weight of $\frac{8}{15}$. Our choice is motivated by the assumption that even though the performance of an autonomous system is relevant during the whole runtime, it is still expected, and consequently accounted for by the system designer, that an untrained system will initially yield suboptimal performance.

$$\begin{aligned} Opt.Metric = &\frac{1}{15} \cdot Perf_{[0:\frac{1}{4}N)} + \frac{2}{15} \cdot Perf_{[\frac{1}{4}N:\frac{2}{4}N)} + \\ &\frac{4}{15} \cdot Perf_{[\frac{2}{4}N:\frac{3}{4}N)} + \frac{8}{15} \cdot Perf_{[\frac{3}{4}N:N)} \end{aligned} \tag{4.7}$$

To obtain valid results quickly, irace has been executed in a parallelized fashion on the nodes of the PC2 compute cluster located at Paderborn

Table 4.1: Value ranges used in the parameter optimization.

Strategy	Parameter	Datatype	Range	
			Lower	Upper
Meta-rules	n	integer	1	1000
	m	integer	1	1000
	e_r	real	0.01	0.99
Global Error	G	real	0.01	10
	W	integer	1	1000
Local Error	G	real	0.01	10
HECS	ΔE_{max}	real	0.01	1
	M_{PA}	real	-10	10
	$L_{exploit}$	real	0.5	1

University[2]. The computational budget assigned to each run of irace depends on both the benchmark environment and the evaluated scenario, which is why the budgets are reported in the corresponding sections.

4.2.3 *Experimental Comparison*

After the optimized parameters have been determined, they have been applied in the benchmark environments to evaluate the performance and behavior of the different strategies. In each environment, 50 repetitions (trials) with different seeds have been conducted. To obtain meaningful results, the seeds differed from those employed in the parameter study. The evaluation was two-fold: First, the optimization metric, as used in the parameter study, has been calculated for each trial and averaged over all repetitions to condense the behavior of each strategy into a single quantity characterizing its performance. The comparison then takes place on a per-environment basis. The methodology outlined by Demšar [24] has been employed to test for statistical significance. First, a Friedman test [31] is applied to identify whether there is a statistically significant difference between the strategies. If this is the case, Holm's step-down procedure [45] is used as post-hoc test with the strategy achieving the best optimization metric as reference. Both tests determine statistical significance at a level of $\alpha = 0.05$.

After the evaluation of the achieved optimization metrics, the behavior of the strategies is inspected visually with graphs showing the development of the performance throughout the experiment. Even when two strategies achieve similar optimization metrics, they might depict behaviors drastically differently. For instance, while one strategy continuously

2 https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua1, accessed 29th March 2023

Table 4.2: Parameterization of the strategies when optimized for each environment separately.

	Meta-rules			Global Error		Local Error	HECS		
	n	m	e_r	G	W	G	ΔE_{max}	M_{PA}	$L_{exploit}$
11-Multipl.	112	17	0.151	6.907	248	2.992	0.013	-8.336	0.988
20-Multipl.	467	36	0.227	0.655	115	0.684	0.380	-2.187	0.935
Maze4	3	1	0.234	1.453	134	7.992	0.800	-9.977	0.999
Woods14	18	1	0.558	9.268	99	9.221	0.527	-0.387	1.000

improves, another could employ pure exploration for a longer time and then swiftly shift to pure exploitation. Even though it cannot be said which behavior is superior, knowing how each strategy behaves is still helpful.

4.3 SINGLE-ENVIRONMENT EVALUATION

This section presents and discusses the experimental results gathered when the parameterization of the strategies is optimized for each environment separately. This evaluation scenario aims at identifying the suitability of the E/E strategies for different problem environments. First, the optimized parameterization of the strategies is discussed in Subsection 4.3.1, followed by the evaluation of the performance of each strategy in each of the four benchmark environments in Subsection 4.3.2.

4.3.1 *Parameter Study*

For the 11-Multiplexer environment, irace has been given a computing budget of 150,000 CPU seconds. Due to the higher problem complexity, the 20-Multiplexer and the Maze4 environment have been optimized with a budget of 600,000 CPU seconds. The highest computing budget of 1,800,000 CPU seconds has been assigned to the Woods14 environment, as it has a larger classifier population and more steps per run than the Maze4 environment.

The parameter configurations resulting from irace's optimization are shown in Table 4.2, where notable differences between the different environments can be observed. For the 20-Multiplexer, the constant number n of exploration runs used in the meta-rules strategy is more than four times higher than for the 11-Multiplexer, indicating the higher complexity of the 20-Multiplexer problem. In the Maze4 and Woods14 environments, n and m are considerably smaller, as the environment employs fewer runs than the two multiplexer environments. The exploration rate e_r of the Woods14 environment is the highest, resulting in a more aggressive adaptation of m. This could be related to the fact that in this multi-step

Table 4.3: Average optimization metrics achieved with the parameters optimized for each environment separately. Bold font marks the best results, an asterisk a statistically significant difference to the best result.

Strategy	11-Multiplexer	20-Multiplexer	Maze4	Woods14
Meta-rules	0.944∗	0.677	4.30	19.67∗
Global Error	0.962∗	0.653∗	4.78∗	14.85
Local Error	**0.972**	0.639∗	7.02∗	81.61∗
HECS	0.969∗	**0.691**	**4.25**	**14.62**

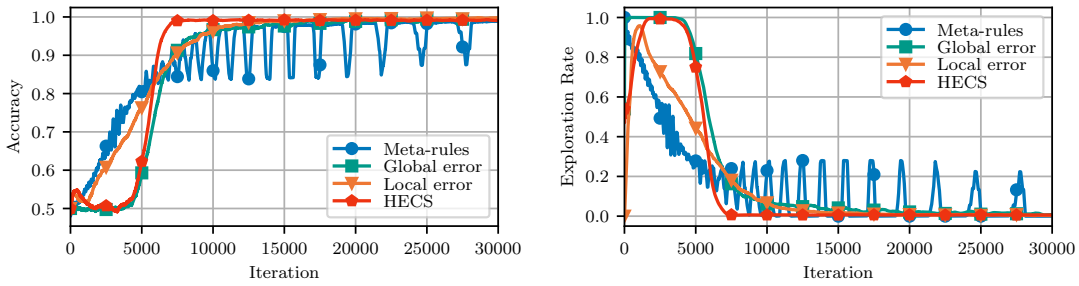
problem, each run consists of up to 100 iterations, allowing XCS to gain more knowledge during a single run.

For the global error strategy, the gain factor G, representing the sensitivity of the exploration probability to the prediction error, is ten times smaller for the 20-Multiplexer than for the 11-Multiplexer. Since we normalize the prediction error to the maximum range of payoffs, the gain factor of 0.655 also represents the upper bound of the exploration probability. For the Woods14 environment with its long action chain, the gain factor of 9.268 is considerably larger than that of 1.453 determined for Maze4. These observations hold roughly for the gain factor of the local error strategy, too, with the only exception that the gain factor for the Maze4 environment is considerably larger, with a value of 7.992.

That the optimal sensitivity of the exploration probability to the prediction error differs between the benchmark environments can also be observed for the HECS strategy, where the M_{PA} parameter is smaller for the 11-Multiplexer than for the 20-Multiplexer. The smaller the value of M_{PA} is, the more sensitive is the accuracy-induced explorer level to the prediction error. Interestingly, the Maze4 environment has the smallest value of M_{PA} and the Woods14 environment the largest, even though it requires a considerably longer chain of actions with small payoff predictions (and errors) in fields distant to the food. One explanation could be that with Woods14, the reward-induced explorer level E_R, which is used to escape unsuccessful exploit trials, plays a more important role than the accuracy-induced explorer level E_A. The maximum change of the explorer level ΔE_{max} is very small and close to zero in the case of the 11-Multiplexer, leading to small changes in the explorer level. For the other three environments, ΔE_{max} is considerably larger. $L_{exploit}$ is close to one for all environments, leading to a low level of minimum exploration and potentially enabling HECS to reach close to optimal performance in all cases.

4.3.2 *Experimental Comparison*

With the optimized parameter configurations applied, the average optimization metrics, as shown in Table 4.3, have been achieved. On the 11-Multiplexer, all strategies achieved values close to 1, showing that



(a) Classification accuracy

(b) Exploration rate

Figure 4.3: Experimental results obtained on the 11-Multiplexer problem. Results are averages of 50 trials and shown as moving average over 400 iterations.

they have been able to emphasize exploitation once the solution to the problem has been learned. The local error strategy achieved statistically significantly better results than the other strategies, even though the magnitude of the differences is small.

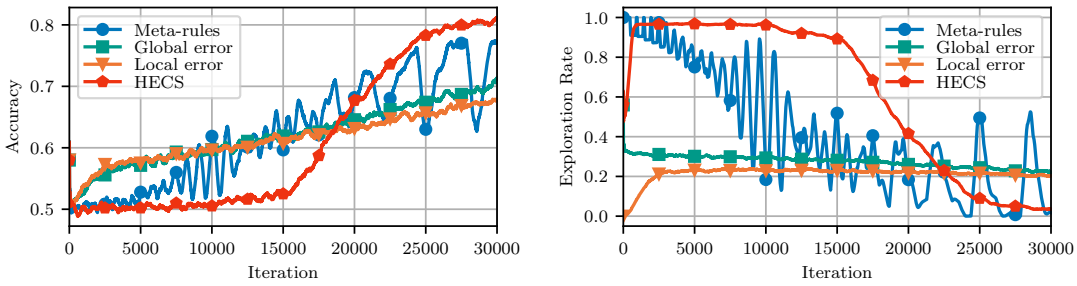
On the 20-Multiplexer, the optimization metrics are considerably smaller, as the problem cannot be solved with the given number of iterations. Here, the HECS strategy takes the lead, but its result is not statistically distinguishable from that of the meta-rules strategy.

In the Maze4 environment, the HECS strategy again achieves the best results and is statistically indistinguishable from the meta-rules strategy. The local error strategy requires considerably more steps, which is also observable in the Woods14 environment. Since the achieved performance metric of approximately 82 steps is close to the maximum of 100 steps, the local error strategy seems to be entirely infeasible for environments with long action chains. HECS needs the fewest steps, followed by the global error strategy with a statistically insignificant difference.

In the second part of the evaluation, we compare how the different strategies behave throughout the experiment, i.e., how the classification accuracy and taken steps, respectively, and the applied rate of exploration develop over time.

11-Multiplexer. Figure 4.3a shows the development of the classification accuracy for all E/E strategies on the 11-Multiplexer. The meta-rules strategy shows a distinct pattern related to the alternating periods of exploration and exploitation. Since the number of explore iterations is not adapted, exploration is applied even when perfect accuracy is achieved during exploitation. The accuracy of the local error strategy is continuously improving until perfect accuracy is reached. On the other hand, both the global error and HECS strategies achieve a very low accuracy at the beginning that suddenly increases after around 5,000 iterations, with the increase of HECS being considerably steeper.

Figure 4.3b shows the corresponding development of the exploration rate. The graph mirrors the development of the accuracy, as the ex-



(a) Classification accuracy

(b) Exploration rate

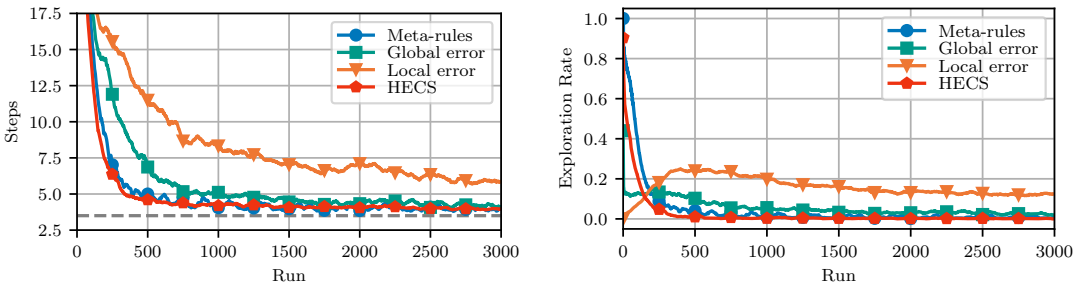
Figure 4.4: Experimental results obtained on the 20-Multiplexer problem. Results are averages of 50 trials and shown as moving average over 400 iterations.

ploration rate of the local error strategy is continuously decreasing. In contrast, the global error strategy first maintains an exploration rate of 1 which suddenly decreases and eventually falls to 0. The HECS strategy behaves similarly, but its exploration rate drops even more quickly. Overall, it seems that the HECS strategy is well suited to determine the turning point when exploitation is called for, while the local error strategy emphasizes a continuous increase in performance.

20-Multiplexer. Figure 4.4a shows the development of the classification accuracy on the more complex 20-Multiplexer. This time, both the global and the local error strategies achieve a continuous increase. The meta-rules strategy is first achieving a worse accuracy but then catches up and outperforms the error-based strategies – at least during its exploitation periods. The HECS strategy depicts a similar behavior than with the 11-Multiplexer problem. At first, the accuracy is not improving, but towards the end, there is a steep increase, eventually leading to the best accuracy in the field.

The development of the exploration rate shown in Figure 4.4b differs from those observed for the 11-Multiplexer. Both error-based strategies start with relatively low exploration rates of 0.2 to 0.3 – a consequence of the small gain factors, which make them more insensitive to high prediction errors. In addition, their exploration rates decrease very slowly. The HECS strategy first applies an exploration rate of 1, which after approximately 15,000 iterations begins decreasing to reach a value close to 0 at the end, mirroring the development of the classification accuracy quite well. However, the 20-Multiplexer is still incompletely solved at the end of the experiment and requires additional exploration afterward to generate a complete solution. In case the operation period of the system is extended, HECS would not apply this and settle on a non-optimal classification accuracy. Hence, the HECS strategy seems overfitted to the specific scenario and the target metric used in the parameter optimization.

Maze4. The development of the steps required to reach the food is shown in Figure 4.5a. The HECS and the meta-rules strategies reach



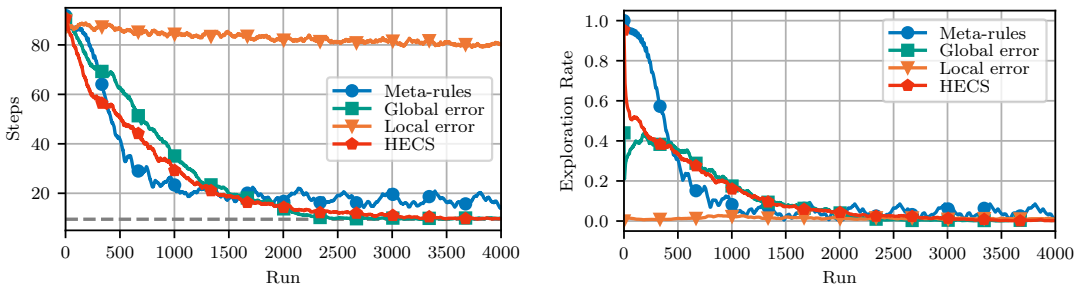
(a) Number of Steps

(b) Exploration rate

Figure 4.5: Experimental results obtained in the Maze4 environment. Results are averages of 50 trials and shown as moving average over 50 runs. The dashed line shows the length of the shortest path and the exploration rate is defined as the percentage of exploration steps in a run.

a close to optimal solution very quickly and show a nearly identical behavior, which aligns with the achieved optimization metrics. An oscillating development cannot be observed for the meta-rules strategy in this visualization, as only $n = 3$ consecutive exploration runs are employed, which are smoothed out by the moving average. The global error strategy requires more runs to reach the optimal solution, while the local error strategy has not achieved this even after 3,000 runs. An inspection of the exploration rates in Figure 4.5b reveals that the HECS and meta-rules strategy both apply a large amount of exploration at the beginning and then quickly reduce it once a (close to) optimal solution is reached. In contrast, the global error strategy reduces the exploration rate immediately to a small amount, which prevents XCS from quickly learning the solution. The bad performance of the local error strategy can be explained similarly by its slow increase in the exploration rate.

We presume that this behavior is related to the specific multi-step characteristics of the maze environment, which makes it challenging for error-based strategies to assess the current capability of the classifier population to solve the problem. In a maze environment, a positive reward is obtained only once at the end of a run and only in case the food is reached. On all other steps, an immediate reward of zero is received. To learn the shortest paths to the food, the reward received upon reaching the food is backpropagated to the preceding classifiers with the Q-learning-like internal reinforcement mechanism of XCS using the discount factor γ. For classifiers early in the path, i.e., those matching positions distant from the food, the path must be repeatedly taken until a small portion of the reward arrives at the classifier. Until then, these classifiers keep a perfectly accurate payoff prediction of zero. However, if most classifiers in the population have a prediction error of zero, the error-based strategies are lured into believing that XCS solves the problem quite well and therefore apply a low exploration rate.



(a) Number of Steps

(b) Exploration rate

Figure 4.6: Experimental results obtained in the Maze14 environment. Results are averages of 50 trials and shown as moving average over 50 runs. The dashed line shows the length of the shortest path and the exploration rate is defined as the percentage of exploration steps in a run.

Woods14. The steps taken in the Woods14 environment, as shown in Figure 4.6a, and the corresponding exploration rates in Figure 4.6b depict this multi-step phenomenon even more distinctly. The HECS strategy and the global error strategy can derive a close-to-optimal solution. Despite showing visible learning progress, the meta-rules strategy cannot achieve a comparable result. On the other hand, the local error strategy is incapable of evolving any viable solution, as it applies close to no exploration. The low prediction error of fields distant from the food leads to a self-reinforcing cycle of non-exploration: Initially, the prediction error is zero, and no exploration is applied when the animat is placed on such fields. However, at an early stage of learning, exploitation will rarely lead to reaching the food, which means that the prediction error for such fields remains low. The global error strategy seems more robust to this problem, as the E/E decision before a run is also affected by the more meaningful prediction errors of classifiers matching to fields close to the food. When combined with the high gain factor determined in the parameter optimization, the global error strategy seems sufficiently sensitive to the global prediction error to apply an adequate amount of exploration.

4.3.3 *Interim Summary*

The HECS strategy, with its accuracy-based exploration level and mechanism to escape unsuccessful multi-step exploitation runs, achieved the best, or close to the best, results in all environments in terms of the optimization metric. However, the development of the exploration rate always depicted a distinct turning point at which the exploration was considerably reduced. This could indicate that its good performance is related mainly to the tailored parameterization and not to the general characteristics of the HECS strategy. The meta-rules strategy also

achieved competitive results in most environments, but since its parameters n and m represent numbers of iterations, their parameterization differs considerably between the environments. The global and local error strategies struggle in our multi-step environments with sparse rewards. However, the global error strategy can compensate for this with a high gain factor G, increasing its sensitivity to the prediction error.

4.4 MULTI-ENVIRONMENT EVALUATION

To test if the performance of the strategies is solely related to their parameters being tailored to each environment individually, we have conducted an experiment in which the parameters of the strategies are not optimized for each environment separately but for all at the same time. Hence, the aim is to determine a "one-fits-all" parameterization of each strategy that is suited to a wide range of different environments.

4.4.1 *Parameter Study*

In order to use irace for optimizing the parameters over all four environments, an optimization metric is required that is comparable over all environments. For the single-step environments, we left the calculation of the target metric (according to Equation 4.7) unchanged. This is not possible for the multi-step maze environments, as they have different value ranges. To normalize them to the same value range used for the single-step environments, we have divided the optimal value of the optimization metric, i.e., 3.5 and 9.5 for Maze4 and Woods14, respectively, by the achieved optimization metric calculated according to Equation 4.7. With that, the target metric for the multi-step problems also lies in the range of 0 to 1, with larger values being favorable. As computational budget, irace has been given 3,150,000 CPU seconds, i.e., the sum of the budgets used in the single-environment optimization.

The resulting parameter configurations are shown in Table 4.4. The meta-rules strategy has very small values of n and m, while the global error strategy has a high gain factor G. This is similar to the parameters observed for the multi-step environments in the single-environment optimization. However, the global error strategy's window size W is larger than in all cases of the single-environment optimization. The local error and HECS strategy have mid-range parameter values compared to the single-environment optimization.

4.4.2 *Experimental Comparison*

The basic behavior of the strategies, e.g., when and how quickly exploration is decreased, has been very similar to the scenario with single-environment parameter optimization, which is why we refrain from a detailed discussion of graphs showing the development of the accuracy

Table 4.4: Parameterization of the strategies when optimized for all environments concurrently.

Meta-rules			Global Error		Local Error	HECS		
n	m	e_r	G	W	G	ΔE_{max}	M_{PA}	$L_{exploit}$
7	2	0.167	9.541	263	2.156	0.599	-2.347	0.984

or exploration. Instead, we restrict our evaluation to the achieved values of the optimization metrics, which are shown in Table 4.5. On the 11-Multiplexer, the local error strategy again achieved the best result, which is only slightly worse than with the single-environment configuration. The same holds for the global error strategy, which achieved the second-best result. The HECS strategy previously achieved the second-best result but now experienced a considerable drop in performance. Again, the exploration rate had a distinct turning point when it switched to exploitation, but in this case, it is too early, so XCS can no longer fully solve this simple benchmark problem. Similarly, the meta-rules strategy also reduces exploration too early. This is likely related to the small number of exploration cycles n and the increase of exploitation once it performs better than during exploration, which is overly optimistic.

On the 20-Multiplexer, the HECS strategy achieves the best optimization metric, and overall the results look similar to the single-environment case, just with reduced values. One exception is the global error strategy, which achieves a value of approximately 0.5, which is the same value that would result from random guessing. Due to its high gain factor G, it is very sensitive to the global prediction error and applies pure exploration the entire time. Hence, this parameterization of the global error strategy seems unsuited to problems that are not completely solvable, as it applies exploitation only once XCS has generated confident payoff predictions.

The relation of the strategies to each other remains unchanged in the Maze4 environment as well. The HECS and meta-rules strategies achieve the best and the second best results at a statistically indistinguishable level, even though the meta-rules strategy maintains a higher exploration rate than in the single-environment case. The optimization metrics achieved by both error-based strategies are more than one step larger, but for opposite reasons. The global error strategy applies exploration for a longer time than in the single-environment case. In contrast, the exploration rate of the local error strategy is close to zero the whole time.

In the Woods14 environment, the behavior of the strategies did not change substantially compared to the single-environment optimization, except that the difference between the results of the meta-rules strategy and the HECS strategy is no longer statistically significant.

Table 4.5: Average optimization metrics achieved with the parameterization optimized for all environments concurrently. Bold font marks the best results, an asterisk a statistically significant difference to the best result. The values in the brackets show the difference to the metrics achieved with the single-environment optimization.

Strategy	11-Multiplexer	20-Multiplexer	Maze4	Woods14
Meta-rules	0.795∗	0.625∗	4.62	21.37
	(−0.149)	(−0.052)	(+0.32)	(+1.70)
Global Error	0.961∗	0.503∗	5.91∗	16.50
	(−0.001)	(−0.150)	(+1.13)	(+1.65)
Local Error	**0.970**	0.608∗	8.67∗	86.71∗
	(−0.002)	(−0.031)	(+1.65)	(+5.10)
HECS	0.892∗	**0.679**	**4.44**	**15.66**
	(−0.077)	(−0.012)	(+0.19)	(+1.04)

4.4.3 *Interim Summary*

Even though the pattern that the HECS strategy seems to perform best also holds in the multi-environment optimization at first glance, our results indicate that no "one-fits-all" parameter configuration exists for any of the strategies. The HECS strategy still achieved the best result in three environments, but it could no longer solve the simple 11-Multiplexer, as it stopped exploration too early. The meta-rules strategy suffered from the same problem but even more distinctively. This is likely related to its parameters n and m, representing numbers of iterations whose magnitudes differ considerably between our single- and multi-step environments. The same holds for the windows size W of the global error strategy. Further, its high gain factor G made the global error strategy infeasible for environments where the complete solution cannot be derived. In both single-step environments, the local error strategy has been affected by only a slight decrease in performance, indicating that it could potentially be the most generally applicable E/E strategy if it would not fail in multi-step environments with sparse rewards.

4.5 DYNAMIC ENVIRONMENT EVALUATION

For true autonomy and adaptivity, E/E strategies must be able to react to environmental changes by applying more exploration to update the evolved solution. So far, all considered environments have been static, and the strategies could start with a high level of exploration and then continuously decrease it. In the scenarios evaluated in this section, the environments undergo a distinct change during the experiment, which forces the strategies to increase exploration to update the classifier population. The parameters are optimized for each environment separately to

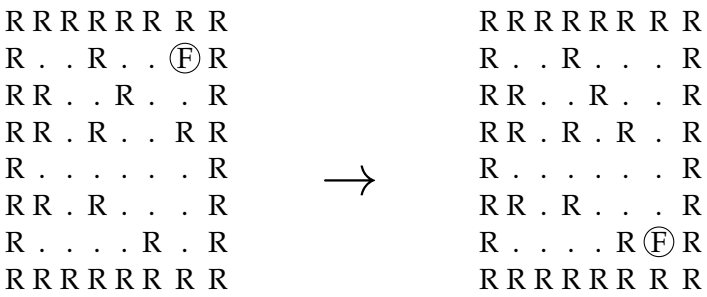


Figure 4.7: The dynamic Maze4 environment. After 3,000 runs, the position of the target field ('F') changes and the experiment continues for another 1,500 runs.

test whether the strategies are, in principle, capable of tackling environmental dynamics. This also allows for a comparison with the parameters found in the single-environment optimization to identify the parameter changes that make a strategy more adaptive.

4.5.1 *Environmental Changes*

To facilitate a comparison with the single-benchmark optimization scenario, the first part of each experiment is identical to the single-benchmark case, followed by a distinct change in the environment and additional iterations to investigate if XCS can react to the change. For both multiplexers, this means that after the initial 30,000 iterations, the two leftmost index bits switch their value for the index calculation. If both bits have the same value, this does not represent a change, but if they have different values, the classifiers that previously matched are no longer accurate. Since the multiplexer bitstrings are randomly sampled with a uniform distribution, 50% of XCS' rule base becomes obsolete after the change, assuming it had previously derived the optimal solution. Therefore, we have increased the number of iterations per experiment by 50%, so each experiment ends after 45,000 iterations.

In the Maze4 environment, the food has been moved from the upper right to the lower right corner, as shown in Figure 4.7. In addition, the rock on the right side has been moved one position to the left to avoid aliasing states, i.e., different fields that lead to the same sensory input making them indistinguishable for XCS. After the change, the average length of the shortest path to the food is approximately 3.58 steps and thus close to the original length of 3.5 steps. In the Woods14 environment, the food has been moved to the other end of the path as shown in Figure 4.8, and the original field of the food, which became empty, has been switched with a neighboring rock to avoid aliasing. In both multi-step environments, the length of the experiments has been extended by 50 % as well since XCS does not have to derive a completely new solution.

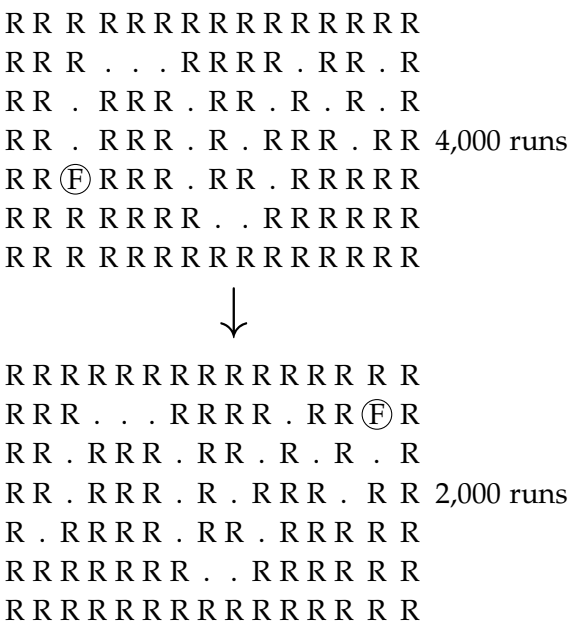


Figure 4.8: The dynamic Woods14 environment. After 4,000 runs, the position of the target field ('F') changes and the experiment continues for another 2,000 runs.

4.5.2 *Parameter Study*

For calculating the optimization metric, the periods before and after the environmental change have been assigned equal weight, i.e., the metric is calculated separately for both periods according to Equation 4.7, and the average of both values represents the final value of the optimization metric. Since the length of the experiments has been extended by 50 %, the computational budget of irace has been increased by the same amount, such that 225,000 CPU seconds have been accounted for the 11-Multiplexer, 900,000 CPU seconds for the 20-Multiplexer and Maze4, and 2,700,000 CPU seconds for Woods14.

The results of the parameter optimization are shown in Table 4.6 along with the parameter changes when compared to the static single-environment optimization. For the meta-rules strategy, the fixed number of exploration runs n is increased in all cases, which seems like an intuitive approach to cope with environmental dynamics. The initial number of exploitation runs m is also increased, but in most cases not as much as the number of exploration runs. The exploration rate e_r is substantially increased for the 11-Multiplexer and the Maze4 environment, nearly unchanged for the 20-Multiplexer, and reduced for Woods14. A reduction of e_r seems counterintuitive for dynamic environments, as large changes to m are necessary to quickly increase the level of exploration, even though this could be offset by the higher number of exploration cycles n.

To make the global error strategy more sensitive to environmental changes, the gain factor G could be increased to make it more sensitive to the prediction error, and the window size W could be reduced to

Table 4.6: Parameterization of the strategies when optimized for each dynamic environment separately. Given in brackets is the difference to the parameters optimized for the static environments.

	Meta-rules			Global Error		Local Error	HECS		
	n	m	e_r	G	W	G	ΔE_{max}	M_{PA}	$L_{exploit}$
11-Multipl.	395	18	0.647	5.044	196	2.982	0.289	−9.995	0.985
	(+283)	(+1)	(+0.496)	(−1.863)	(−52)	(−0.010)	(+0.276)	(−1.659)	(−0.003)
20-Multipl.	618	84	0.196	2.122	150	3.850	0.770	−3.885	0.970
	(+151)	(+48)	(−0.031)	(+1.467)	(+35)	(+3.166)	(+0.390)	(−1.698)	(−0.035)
Maze4	21	29	0.790	1.113	53	7.524	0.916	−6.400	0.993
	(+18)	(+28)	(+0.556)	(−0.340)	(−81)	(−0.468)	(+0.116)	(+3.577)	(−0.006)
Woods14	22	1	0.338	9.673	31	9.619	0.220	4.413	1.000
	(+4)	(±0)	(−0.220)	(+0.405)	(−68)	(+0.398)	(−0.307)	(+4.800)	(±0.000)

make the estimate of the prediction error react more quickly. Of all four environments, only the Woods14 environment shows such parameter changes. In the 11-Multiplexer and the Maze4 environment, W is reduced while, at the same time, G is decreased. The opposite happens for the 20-Multiplexer, where both G and W are increased.

Since the E/E decision of the local error strategy solely depends on the prediction errors of the classifiers matching the current input, it is the only strategy with no internal state or momentum. As such, it has the potential to cope with environmental dynamics with the same parameterization as in the static case. And indeed, in the 11-Multiplexer, virtually no change of the gain factor G is observed, and in the Maze4 and Woods14 environment only small changes. On the other hand, a rather drastic increase is seen on the 20-Multiplexer, where the gain factor for the static case has been relatively small. This is likely related to the inability to derive a complete solution for the 20-Multiplexer, which, in combination with the chosen optimization metric, forced the local error strategy to apply only a small amount of exploration in the static environment, which no longer seems to be optimal for the dynamic case with its additional iterations.

The intuitive approaches to make the HECS strategy more suitable for dynamic environments encompass a reduction of $L_{exploit}$ to increase the minimum base amount of exploration, a reduction of M_{PA} to increase the sensitivity to the prediction error, and an increase of ΔE_{max} to increase the magnitude of changes to the explorer level and react to environmental changes more forcefully. In all four environments, $L_{exploit}$ is nearly unchanged, showing that it is still possible to reach close to pure exploitation. For both multiplexers, M_{PA} is reduced, and ΔE_{max} is increased. The latter is also the case for the Maze4 environment, but M_{PA} is increased here. For the Woods14 environment, ΔE_{max} and M_{PA} developed in the opposite direction as predicted. One possible explanation for this is that the primary source of exploration in such a multi-step

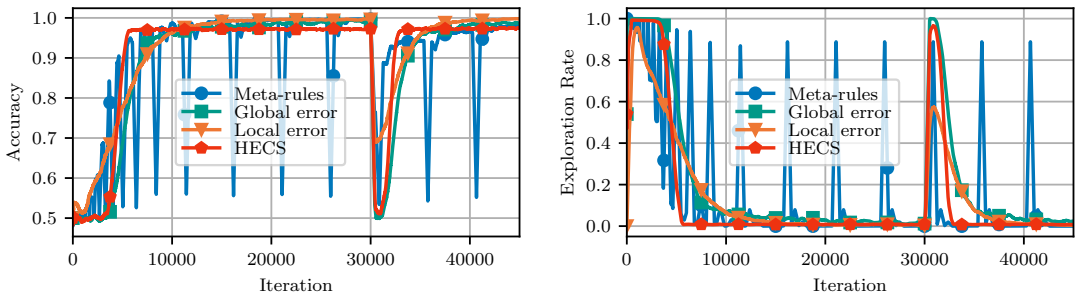
Table 4.7: Average optimization metrics achieved with the parameterization optimized for each dynamic environment separately. Bold font marks the best results, an asterisk a statistically significant difference to the best result. The values in the brackets show the difference to the metrics achieved in the static environments.

Strategy	11-Multiplexer	20-Multiplexer	Maze4	Woods14
Meta-rules	0.940∗	0.716∗	5.15∗	16.72∗
	(−0.004)	(+0.039)	(+0.85)	(−2.95)
Global Error	0.961∗	0.696∗	5.33∗	15.70
	(−0.001)	(+0.043)	(+0.55)	(+0.85)
Local Error	**0.975**	0.680∗	6.46∗	85.10∗
	(+0.003)	(+0.041)	(−0.56)	(+3.49)
HECS	0.956∗	**0.741**	**4.42**	**13.60**
	(−0.013)	(+0.050)	(+0.17)	(−1.02)

environment with a long chain of actions is not the accuracy-induced but the reward-induced explorer level, which is used to escape unsuccessful exploitation trials and is not influenced by the optimized parameters.

4.5.3 *Experimental Comparison*

The achieved optimization metrics are shown in Table 4.7, along with the difference to the results achieved in the static single-environment optimization case. Even though the metrics achieved in the static environments are not directly comparable to those of the dynamic environments, the observed difference still allows for assessing which strategies cope better with dynamics. Overall, the results show a pattern similar to the static case, as the local error strategy performs best on the 11-Multiplexer and HECS on the other benchmarks. Further, no strategy has been affected by a considerable drop in the optimization metric, indicating that all strategies are, in principle, able to sustain their performance when faced with environmental dynamics – at least with optimized parameters. On the 11-Multiplexer, all strategies observed a (slight) decrease in the target metric, except for the local error strategy, which improved its achieved optimization metric even further. On the 20-Multiplexer, all strategies improved when compared to the static case, which can be explained by the inability to fully solve the benchmark in the static scenario and the additional iterations of the dynamic scenario providing further learning opportunities. In the Maze4 environment, the performance of all strategies decreased except for the local error strategy. In the Woods14 environment, results are mixed, as the Meta-rules and HECS strategy showed improvements over the static case. In contrast, both error-based strategies have been affected by a decrease in the optimization metric.



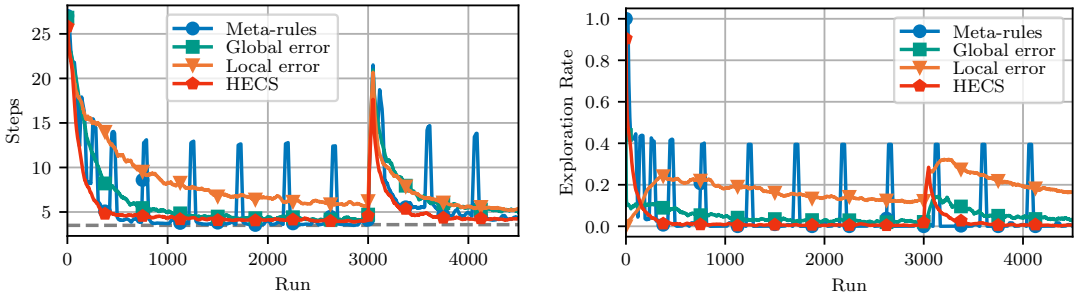
(a) Classification accuracy

(b) Exploration rate

Figure 4.9: Experimental results obtained on the dynamic 11-Multiplexer problem. Results are averages of 50 trials and shown as a moving average over 400 iterations.

While the results of the Meta-rules and HECS strategy are mixed, the global and local error strategies show a more distinct pattern in the presence of environmental dynamics. The global error strategy has improved its performance only on the 20-Multiplexer, while it showed decreasing or close to constant performance in all other environments. On the other hand, the local error strategy improved in all environments except Woods14, where it still not yields reasonable results. On the 11-Multiplexer and Maze4, it even was the only strategy to accomplish an improvement. Overall, the comparison between the local and global error strategy could indicate that local E/E decisions are better suited to cope with environmental dynamics, presumably because only with local decisions is it possible to distinguish between environmental niches that changed and those that did not and thus do not require exploration.

This can be well observed in Figures 4.9 and 4.10, showing the development of the experiments for the 11-Multiplexer and the Maze4 environment, respectively. After the change of the 11-Multiplexer at iteration 30,000, the local error strategy increases the exploration rate to a value just above 0.5, corresponding to the fraction of environmental niches that undergo a change and require new rules. All other strategies have applied a higher exploration rate. This is also reflected in the classification accuracy, where the local error strategy experienced the smallest drop in accuracy after the change. In the Maze4 environment, the global error strategy requires considerably fewer steps than the local error strategy before the food is moved at run 3,000. However, after this point, both strategies achieve very similar results, showing that, relative to the local error strategy, the global error strategy struggles to incorporate the environmental change. Detailed results of the 20-Multiplexer and Woods14 environments are not shown, as all strategies show a behavior very similar to the static environment scenario – both before and after the change.



(a) Number of Steps

(b) Exploration rate

Figure 4.10: Experimental results obtained in the dynamic Maze4 environment. Results are averages of 50 trials and shown as a moving average over 50 runs. The dashed line shows the length of the shortest path and the exploration rate is defined as the percentage of exploration steps in a run.

4.5.4 *Interim Summary*

Our evaluation in environments that undergo a distinct change revealed that all strategies are able to cope with such environmental changes, at least when their parameters are specifically optimized for such a case. Intuitively, all strategies except the local error strategy have several parameters which could be adjusted to increase adaptivity. However, our parameter optimization revealed that not all parameters always develop in the predicted direction, which could indicate that manually setting parameters for dynamic environments can be challenging.

Further, our evaluation results indicate that local E/E strategies can have an advantage over global approaches when not all niches of the environment are affected by changes simultaneously. With local decisions, exploration can be specifically applied to niches for which the classifiers must be updated, while exploitation can be kept in unchanged niches, which preserves operational performance.

4.6 SUMMARY AND DEPLOYMENT GUIDELINES

We have evaluated four different E/E strategies. The HECS strategy, with its non-linear and accuracy-based working mechanism, has achieved the best results in most cases, at least according to the target metric employed during parameter optimization. However, it often achieved this with a distinct turning point at which a sudden switch from full exploration to exploitation is made. In the case of a non-optimal parameter configuration, it was observed that this could lead to XCS settling on an unnecessarily low level of performance, even in a simple learning environment. Overall, the HECS strategy seems to be rather sensitive to non-optimal parameters. The same holds for the meta-rules strategy, whose main parameters heavily depend on the number of iterations

that XCS performs in the problem environment. Its oscillating behavior is a result of a fixed number of exploration iterations. The shift from exploration to exploitation, once the latter achieves better performance, can be overly optimistic if parameterized inappropriately.

Both the global and local error-based strategies are based on the error of the payoff predictions of the classifiers in XCS, which results in a tendency of decreasing exploration more steadily than the HECS and meta-rules strategies, especially in the case of the local error strategy. However, taking the error as the decision criterion reaches its limitation in multi-step environments with sparse rewards. While the global error strategy can compensate for this with an appropriate parameterization, the local error strategy cannot evolve any feasible solution in an environment that requires learning a long chain of actions. In the face of environmental dynamics, on the other hand, the local error strategy demonstrated that it applies exploration only in states that have changed – an ability unique to local E/E approaches.

Even though our instrumental study has investigated only artificial learning problems commonly used in LCS research, it is still possible to derive preliminary guidelines for deploying E/E strategies in real-world problem environments. The key findings relevant for XCS practitioners concern the strategy selection and which factors must be accounted for when configuring the strategies' parameters.

Parameterization. In terms of the parameterization of the strategies, a system designer seems to need to consider three major factors, namely (a) if XCS is, in principle, able to solve the problem environment in the envisioned time frame completely, (b) if the environment is static or undergoes changes, and (c) if the environment is a single-step or a multi-step environment.

Strategy Selection. Overall, we deem the local error-based strategy most promising for XCS practitioners. It results in a steady shift from exploration to exploitation and is, at least in single-step environments, resistant to the problem of stopping exploration too early, as it is the case for the HECS strategy. Furthermore, of all evaluated strategies, it is the easiest to parameterize, as the sensitivity to the prediction error is the only parameter and can be the same for static and dynamic environments. Further, local E/E strategies have the ability to apply exploration only in states where it is necessary, which is beneficial especially in dynamic environments. This property could also be helpful in environments with state imbalances [77], where exploration in frequently occurring states can be reduced earlier. However, all advantages mentioned above only apply to single-step problems. The local error-based strategy is not suited for multi-step environments with sparse rewards, where the prediction error in states distant from the target state is initially not representative of the learning progress of XCS.

4.7 CONCLUSION AND FUTURE WORK

This chapter presented an experimental evaluation of four different E/E strategies for XCS both in single- and multi-step environments. A comparison of a local error-based strategy with three global approaches has yielded distinct differences in their behavior and operational performance in different environments, thereby providing new insights for the development of E/E strategies that are necessary to make XCS suited for deployment in autonomous self-aware systems. In addition, a systematic automated parameter optimization has been conducted both in static and dynamic environments to find reasonable parameter configurations and investigate the parameter sensitivity of the evaluated strategies. Even though we have focused on XCS, our findings could also be representative of other variants of Michigan-style LCSs as long as they use an accuracy-based fitness evaluation.

Since we deem the local error-based strategy as the most promising, future work could try to equip it with escape mechanisms tailored specifically to multi-step environments with sparse rewards, akin to the approach used in the HECS strategy. For instance, the local error-based E/E decision could be considered only in states where the prediction error already has a representative meaning. In terms of improving the universality and reliability of all evaluated strategies, their parameter sensitivity could be reduced by adapting the parameters at run-time to suitable values, e.g., by deriving them from performance or population state metrics [52]. Further, our evaluation has not considered the action selection during exploration, as all evaluated strategies rely on random action selection. However, a directed action selection could improve exploration efficiency and, therefore, needs to be investigated by future work.

5

SAFETY GUARANTEES THROUGH FORBIDDEN CLASSIFIERS

Concepts of computational self-awareness are increasingly being applied when designing self-adaptive Cyber-Physical Systems (CPSs). Such systems depict several distinct challenges, of which many have yet to be addressed [9]. One of the most important ones is to design CPSs that depict a high degree of autonomy and adaptivity but still exhibit safety guarantees to prevent catastrophes during operation. Since a CPS has a real-world impact, it can permanently harm itself, its environment, or living beings in its surrounding. However, as online learning mechanisms are inherently explorative, a self-aware CPS might execute such harmful actions purely out of curiosity. Proper means to prevent such safety-critical events are part of current research in the field of machine learning and artificial intelligence [4].

When XCS and other Reinforcement Learning (RL) techniques are used to implement self-* properties, they operate without supervision, as continuous runtime monitoring is infeasible or at least undesired. Since detailed apriori knowledge of the operational environment is lacking, the learning techniques must be inherently adaptive to adjust to the environmental characteristics. In the RL paradigm, the system learns which behavior is beneficial by trying different actions and observing the feedback from the environment. As such, RL techniques are prone to violate safety requirements during operation since adapting to the environment encompasses an exploration of the action space. Especially when Explore/Exploit (E/E) strategies with a minimum amount of exploration are employed, such as meta-rules (cf. Chapter 4), safety violations during the system's operation are inevitable.

Existing work in the field of safe RL often relies on an additional subsystem external to the learning algorithm. The subsystem is equipped with pre-defined knowledge and constantly monitors the situation and the learner's behavior to intervene whenever harmful actions are about to be executed [32]. In this chapter, we focus specifically on XCS and propose an approach in which the safety-critical knowledge is directly embedded into its knowledge base, making an external monitor superfluous. To achieve this, we use the interpretability of the rules inside XCS and introduce the concept of *forbidden classifiers*, which are rules that do not propose an action but prevent it from being executed in safety-critical situations. So far, the introduction of safety guarantees has rarely

been discussed in the context of Learning Classifier Systems (LCSs) or XCS specifically. To the best of our knowledge, this work is among the first to leverage the interpretability of XCS' rule base by systematically introducing hand-crafted classifiers.

Parts of this chapter originally appeared at the EvoApplications conference 2022 [38] and received a best paper award nomination. An invited extension is currently under review by the Springer Nature Computer Science journal [35]. During his time as a student research assistant, Mathis Brede contributed to this work by extending the C++ XCS library developed in his Bachelor's thesis with forbidden classifiers and implementing the experimental setup. In summary, this chapter contributes to the existing body of research in the following aspects:

- Forbidden classifiers integrate safety-critical domain knowledge directly into the classifier population and spare XCS from learning this knowledge through interactions with an external shield. To the best of our knowledge, the proposed concept of forbidden classifiers is among the first to systematically insert domain knowledge into XCS's classifier population.
- We experimentally quantify the advantage of forbidden classifiers over a safety monitor external to XCS. It is shown that XCS with forbidden classifiers can evolve a problem solution in a shorter time, with a smaller classifier population and a lower computational burden.
- In addition, a supervised classification task is evaluated, demonstrating that, even in the absence of safety requirements, XCS benefits from domain knowledge that is manually injected through forbidden classifiers.

The chapter continues with Section 5.1 by outlining related work in the field of safe RL. The algorithmic modifications necessary for implementing forbidden classifiers in XCS are presented in Section 5.2. Section 5.3 describes our experimental setup. The first results are discussed in Section 5.4, where the impact of forbidden classifiers on the learning mechanism of XCS is investigated for a simple but well-interpretable multiplexer problem. The main experiment is presented and discussed in Section 5.5, comparing the use of forbidden classifiers to XCS with an external safety monitor in three different maze navigation environments. As last experiment, the potential benefits forbidden classifiers offer in classifying a dataset from the UCI repository are investigated in Section 5.6. Finally, Section 5.7 concludes the article and outlines future work.

5.1 RELATED WORK

According to a survey by García and Fernández [32], the approaches to introduce safety constraints into RL techniques can be categorized either as

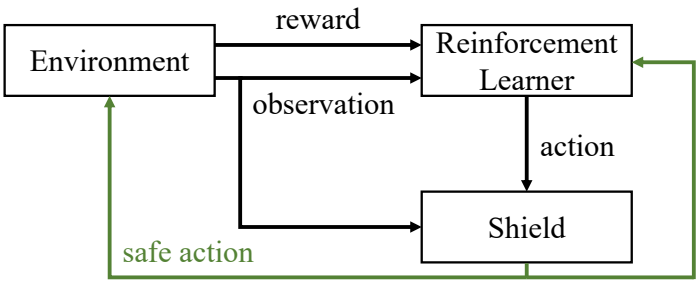


Figure 5.1: Shielded reinforcement learner [3].

a modification of the optimization criterion or of the exploration process. Techniques from the first category do not solely focus on maximizing the payoff received from the environment, as most standard RL algorithms do, but also take into account some measure of risk, e.g., the variance of the received payoff. However, these approaches are often overly pessimistic, leading to non-optimal problem solutions. Further, they only improve the safety level if reaching a catastrophic state increases the risk measure, which is not necessarily the case for arbitrary problem domains. Further, such approaches only reduce the probability of the occurrence of risky situations but do not give any strict guarantees.

Entirely avoiding catastrophic states from the beginning of operation is only possible if external knowledge is used. Otherwise, the only way for a learning system to identify harmful actions is to try them out at least once. Techniques that belong to the second category of safe RL approaches and modify the exploration process make use of this. The external knowledge can either be inserted into the system via (1) a pre-deployment learning phase with manually created samples and policies or (2) through teacher advice at runtime. The first approach does not require an additional subsystem to be deployed in the final system. However, it cannot give strict guarantees because the exploratory nature of online learning can still lead to catastrophic states during operation. On the other hand, a constantly active teacher can give such strict guarantees, at least if the teacher-learner relationship is designed such that safety-critical advice is mandatory for the learner. The teacher can either be formally specified by the system designer, a human in the loop, or even a learner by itself that was trained to imitate human intervention [88].

Closely related to the concept of teacher-advised RL is the approach of shielded learning, as proposed by Alshiekh et al. [3] and shown in Figure 5.1. The shield constantly monitors the state of the environment and the action that the learner is proposing. Whenever an action is proposed that is considered harmful in the current state of the environment, the shield overrides the action to prevent catastrophic consequences. In [3], a formal method with proven correctness and minimal interference is presented to synthesize a shield for Markovian Decision Problems (MDP) based on safety requirements given as temporal logic specification. Exper-

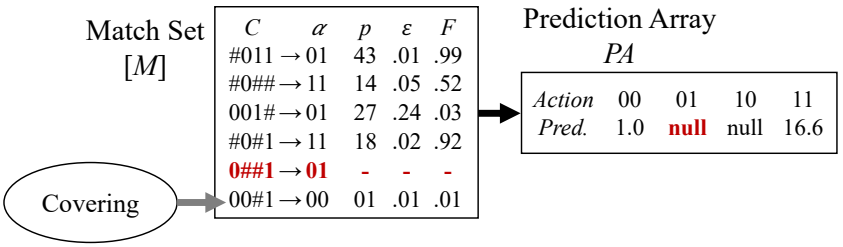


Figure 5.2: The same situation as already shown in Figure 3.2, but with a forbidden classifier (marked red) that is preventing the selection of the action 01. θ_{mna} is still set to 2, so a new classifier is created via covering for one of the actions that have not yet been part of PA.

imental results indicate that a shield not only provides safety guarantees but also improves the speed of learning.

For XCS specifically, Tomforde et al. [104] proposed a teacher-like approach in which newly created classifiers must pass a safety check in a simulation environment before being inserted into the population. The functionality of XCS is split into two layers. The first layer is responsible for selecting an action based on the current situation and the classifiers present in the population. It only performs parameter updates and no generation of new classifiers. This does not only imply the absence of a Genetic Algorithm (GA) but also of a covering mechanism. If no classifiers match the current situation, a classifier is not randomly created via covering. Instead, the action of a classifier whose condition is similar to the current input is selected. The similarity between a condition and the input is measured with a configurable distance metric. The generation of new rules occurs in the second layer, where the classifiers generated by a GA are evaluated in a simulation before being inserted into the population. Since the simulated environment is assumed to be sufficiently accurate, it is possible to detect classifiers that violate safety requirements and discard them before inserting them into the population.

In contrast to the described approaches, our method differs in how the safety-critical knowledge integrates into the system. While both teacher and shielding approaches require an additional subsystem that is external to the learning algorithm and constantly monitors it, we insert the knowledge directly into the learning base of XCS with manually created rules and ensure the safety guarantees through minor algorithmic modifications without the need to learn from interactions with an external subsystem.

5.2 FORBIDDEN CLASSIFIERS

To prevent XCS from taking catastrophic actions, we leverage the interpretability of the classifiers given by their condition→action rule structure and propose the use of rules that *forbid* an action in certain situations instead of proposing it. Consequently, we term this kind of XCS rules

forbidden classifiers. Such classifiers model situations where a specific action is harmful or has catastrophic consequences and should not be executed. In order to extend XCS with the ability to exhibit safety guarantees via the integration of forbidden classifiers, the following algorithmic modifications have been employed:

1. Forbidden classifiers are inserted into the classifier population $[P]$ upon initialization of XCS. The numerosity n of a forbidden classifier is set to the constant value of 1, representing the minimal value that the numerosity can take. The remaining parameters of a forbidden classifier, e.g., the payoff prediction p, are of no further relevance due to the prohibitive nature of the classifier.

2. Whenever a forbidden classifier is part of the match set $[M]$, the action that it forbids is excluded from the prediction array PA, regardless of other classifiers in $[M]$ that might propose it. This assures that in safety-critical situations, the action of a forbidden classifier is never considered during action selection. If less than θ_{mna} actions are present in the prediction array PA, covering is applied until either θ_{mna} actions are part of PA, as shown in the example given in Figure 5.2, or no additional actions are available.

3. Forbidden classifiers do not participate in the GA. Considering that their sole purpose is to prevent the execution of harmful actions, there exists no need to evolve new classifiers from them. In the most common variant of XCS, participation in the GA is prevented by design, as the GA works on the action set $[A]$ that forbidden classifiers are never part of.

4. XCS keeps a list of all forbidden classifiers to check if an offspring classifier generated by the GA is subsumed by one of the forbidden classifiers. If this is the case, it is not inserted into $[P]$. This additional subsumption mechanism is necessary because the existing subsumption mechanisms work solely on the action set $[A]$, in which a classifier that is subsumed by a forbidden classifier will never be present. This subsumption must be checked only when the GA has mutated the action of the offspring. The GA operates on the action set, in which all classifiers have the same non-forbidden action. It assures that the generated offspring still matches the current input, which guarantees that the offspring always matches at least one non-forbidden state/action pair as long as its action remains unchanged.

   It is noteworthy that this subsumption mechanism is not able to detect if a combination of multiple forbidden classifiers overlays an offspring classifier. Possible solutions to this problem, such as an enumeration of all matching states of the offspring classifier followed by a comparison with the matching states of all forbidden classifiers, are potentially compute-intensive. Hence, we refrained

from their usage. If an offspring is inserted into the classifier population that is entirely overlaid by forbidden classifiers, its action is never chosen for execution and eventually gets deleted. Since its experience exp stays at zero, its deletion vote will remain low, meaning it might take some time to get deleted. During this time, it occupies space in the classifier population, which could instead be used for classifiers that could contribute to the problem solution. As such, the unnecessary insertion of offspring classifiers that are fully overlaid by forbidden classifiers does not have any direct detrimental effects, but it could make the learning mechanism of XCS less efficient.

5. Forbidden classifiers are made transparent to the deletion mechanism, i.e., they do not give a deletion vote during the roulette-wheel selection. Due to their non-participation in the selection, forbidden classifiers are never deleted from the population, preserving the safety guarantees they represent.

Overall, forbidden classifiers can be characterized as static and passive, as they are never deleted, do not participate in the GA, and their action is never selected for execution. As such, the concept of forbidden classifiers can be incorporated not only into XCS but into all learning classifier systems that depict a condition→action structure, e.g., into XCSR with its interval-based real-valued conditions [97] or XCS with code-fragment based conditions [47]. The classifiers required to fulfill given safety guarantees can either be created manually by the system designer using domain knowledge or even automatically, e.g., using the approach for shield synthesis proposed in [3]. However, the latter approach requires an MDP model of the environment, which for some environments is not necessary, as the required forbidden classifiers can easily be handcrafted. In any case, it must be assured that no forbidden classifier is entered incorrectly and that in all situations, at least one action remains for XCS to execute.

Even though using static rules based on domain knowledge to prevent harmful actions from being executed is relatively straightforward and closely related to the concept of an external shield, we still argue that it is favorable to internalize such knowledge into the learning mechanism. With an external shield, the safety-critical knowledge embedded into it must still be internalized through learning, potentially wasting time and computing resources. By manually inserting correctly generalized forbidden classifiers, XCS is relieved from the burden of finding adequate generalizations, at least in the safety-critical niches. If adequate generalizations of the forbidden states are straightforwardly derived from the safety requirements, the forbidden classifiers can be easily specified by hand. However, the problem of determining the minimal number of forbidden classifiers that cover all forbidden states is equivalent to the minimization of two-level Boolean logic minimization, which is generally an NP-complete task [105]. If proper generalizations cannot be derived

by hand, one of the well-researched optimization heuristics, e.g., from the ESPRESSO family [12], could be employed to find close to a minimal number of forbidden classifiers in a reasonable amount of time. Thus, forbidden classifiers promise to incorporate the safety guarantees into the classifier population minimally intrusively without "distracting" XCS from the actual problem-solving.

5.3 EXPERIMENTAL SETUP

In order to investigate if forbidden classifiers can hold this promise, our experimental evaluation is mainly guided by two questions:

1. How do forbidden classifiers impact the learning process of XCS, especially that of the GA?
2. Compared to the shielding approach, does the insertion of forbidden classifiers provide XCS with any advantage?

Tackling question (1), we first consider the 6-Multiplexer problem in Section 5.4. The 6-Multiplexer is a trivial binary classification problem that does not require any safety guarantees. However, it allows for investigating the effects of forbidden classifiers on the learning mechanisms inside XCS, as the classifier population can easily be interpreted and evaluated because of the small input and action space of the 6-Multiplexer.

The main part of our experimental evaluation concerns question (2) and takes place in Section 5.5, where the use of forbidden classifiers is evaluated in three different maze environments, in which XCS is navigating a robot to find a target field with a minimal number of steps. As part of such robot navigation tasks, it must, under all circumstances, be avoided that the robot crashes into an obstacle to avoid damage to itself and its environment. We guarantee this by inserting appropriate forbidden classifiers and compare our approach to XCS with an external shield.

The shielding approach presented in [3] assumes the learner can update multiple policies in parallel. Since no straightforward way exists to implement this for XCS, different approaches to realizing an external shield are thinkable. While parameter updates on multiple action sets in parallel are possible, it is unclear whether the GA executed on one action set should insert its offspring classifiers into the other action sets in case they match. If so, the order in which the GA is executed on the different action sets might affect the learning progress. In addition to such open questions, a required modification of XCS eradicates the main advantage of an external shield, which is its agnosticism to the learning algorithm. Hence, we opted for a simpler shield that provides the smallest, or most negative, reward possible in the environment every time XCS chooses an unsafe action. Afterward, XCS is called again until a valid action is chosen, i.e., all parameter updates are fully sequential, and the shield cannot force XCS to select a particular action. Since parameter updates

occur both during exploration and exploitation, it is guaranteed that, eventually, a valid action is selected by XCS on its own. With the additional XCS invocations imposed by the shield, XCS is given additional learning opportunities to internalize the knowledge embedded in the shield without requiring any modifications to XCS.

As an additional use case outside of safety-critical RL applications, we investigate the classification of a dataset from the UCI repository [28] in Section 5.6. In this case, the injected forbidden classifiers do not implement safety guarantees but represent human knowledge inserted to bootstrap the classifier population and improve its (initial) classification accuracy.

For our experiments, we have employed our C++ implementation[1] of XCS, which follows the algorithmic specification given in [17]. Just as in Chapter 4, the specify operator of Lanzi [56] has been employed in the maze environments to tackle overgeneralization. The maximum population sizes N, the discount factor γ, and the specify parameters n_{sp} and p_{sp} have been tailored to each environment. Apart from that, a common parameter configuration of XCS has been used.[2] Explore and exploit trials have been strictly alternating. Since the operating performance during exploration is not allowing any conclusions about the state of learning, only the performance during exploitation trials is reported. To obtain meaningful results, each experiment has been repeated 100 times with different random seeds, with the problem environments and XCS using separate random number generators. Execution times have been measured with the `std::clock` utility and performance and population tracking disabled. All experiments have been conducted on an off-the-shelf desktop computer with an Intel i7-7700K CPU, 16 GiB RAM, and Debian Linux 11. When measuring execution times, no other user programs have been running in parallel.

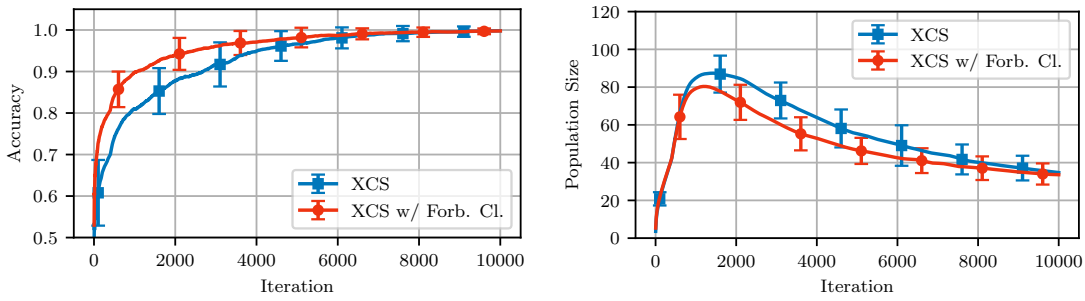
5.4 EXPERIMENTAL EVALUATION: 6-MULTIPLEXER

The 6-Multiplexer is the smallest and simplest representative of the multiplexer problem environments, which have been described in Section 4.2. Its simplicity allows for an effortless inspection of the evolved classifier populations. For the 6-Multiplexer, the optimal population consists of 16 classifiers, each with three bits specified – the two index bits and the one they point to. Overall, this results in eight classifiers that propose the correct action and eight classifiers proposing the wrong action, which must also be present in the population since XCS learns to predict the reward for all state/action pairs accurately.

To investigate the effect of forbidden classifiers on the learning mechanism of XCS, we have initialized the population with the two forbidden

1 https://git.uni-paderborn.de/xcs/xcs-safety, accessed 31.03.2023

2 That is $\beta = 0.2$, $\alpha = 0.1$, $\nu = 5$, $\mu = 0.04$, $\delta = 0.1$, $p_I = 10$, $\epsilon_I = 0$, $f_I = 0.01$, $P_\# = 0.5$, $\epsilon_0 = 10$, $\chi = 0.8$, $\theta_{GA} = 25$, $\theta_{sub} = 20$, $\theta_{del} = 20$, $DoGaSubsumption = True$, $DoActionSetSubsumption = False$.



(a) Classification accuracy

(b) Population size

Figure 5.3: Experimental results obtained on the 6-Multiplexer problem. Results are averages over 100 trials and shown as a moving average over 200 samples. The error bars visualize the observed standard deviation.

classifiers 001###→0 and 10##1#→0, which represent the case that either the first or the third bit is indexed and has the value 1, but action 0 is selected. Since forbidding the wrong action in a binary classification problem inevitably leads to selecting the correct action, these two forbidden classifiers already provide the classifier population with 25% of the problem solution. Therefore, with random guessing as the baseline, it can be expected that XCS with forbidden classifiers achieves a classification accuracy of about 12.5 percent points better than XCS alone.

Figure 5.3a shows the development of the classification accuracy both for the standard XCS and XCS with added forbidden classifiers. The maximum population size N has been set to 200, and 10,000 iterations have been performed overall. At the beginning of the experiment, the advantage of XCS with the two forbidden classifiers is indeed around 12.5 percent points but is reducing quickly in the following iterations as the standard XCS is learning the knowledge represented by the forbidden classifiers as well. At the end of the experiment, both variants of XCS reliably achieve perfect classification accuracy. The size of the macro-classifier population, i.e., the number of distinct classifiers in the population not considering their numerosities n, develops as shown in Figure 5.3b. With forbidden classifiers, the population is smaller, indicating better generalization since the problem can be solved with fewer classifiers. Both population sizes seem to converge towards the end of the experiment, showing that a solution of similar quality can be evolved without forbidden classifiers as long as enough learning iterations are applied.

The reason for the improved generalization capabilities becomes apparent when manually inspecting the evolved classifier populations. Figure 5.4 shows an excerpt of a population that has been evolved through one of the trials. Most distinctly, the classifier #0####→0 is among the classifiers with the highest numerosity n and predicts the correct payoff of 1,000 with a prediction error ϵ of 0, i.e., with perfect accuracy. Normally, it would be considered overgeneralized since classifiers that

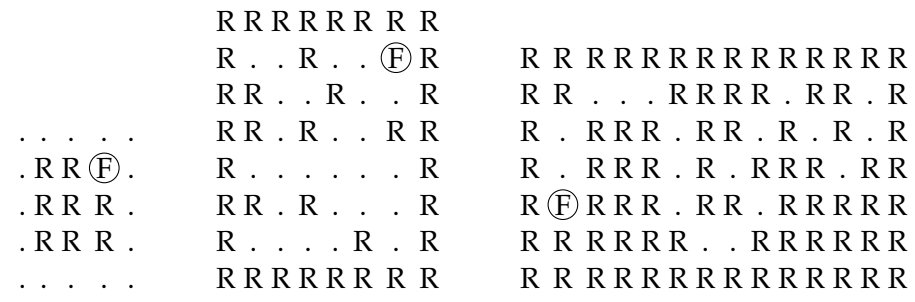
C	α	n	p	ϵ	F
#0#### $\rightarrow$	0	19	1000	0	0.825
01#1## $\rightarrow$	0	19	0	0	0.999
1####0 $\rightarrow$	0	19	1000	0	0.925
10##1# $\rightarrow$	1	17	1000	0	0.942
0##0## $\rightarrow$	0	17	1000	0	0.812
001### $\rightarrow$	1	15	1000	0	0.965
		...			
001### $\rightarrow$	0	1	-	-	-
10##1# $\rightarrow$	0	1	-	-	-

Figure 5.4: Excerpt of an exemplary classifier population that has been evolved by the end of a trial. The classifiers are sorted according to their numerosity n in descending order. Forbidden classifiers are marked red.

predict the correct payoff in the 6-Multiplexer problem can, at maximum, be generalized to have 3 bits specified. The only specified bit is the second index bit, meaning that in matching situations, the bit to predict is either the first or third bit of the remaining bits. However, if one of these bits is indexed and has the value 1, one of the forbidden classifiers matches, and choosing the action 0 is prevented. Hence, whenever the wrong action is proposed, the forbidden classifiers become active and prevent its action from being selected. This also applies to all other accurate classifiers with less than three specified bits. The mechanism of forbidding actions in specific niches of the environment thus enables XCS to generalize classifiers into the niches of forbidden classifiers, potentially reducing the population size. Hence, the introduction of forbidden classifiers not only acts as a filter for the action selection but can aid XCS' learning process by opening generalization possibilities that normally do not exist.

5.5 EXPERIMENTAL EVALUATION: MAZE

Maze environments require XCS to use its multi-step learning capabilities and have already been presented in Section 4.2. The three different environments employed for the experimental evaluation of forbidden classifiers are shown in Figure 5.5. They depict different complexities, as they differ not only in the number of empty fields and rocks but also in the length of the shortest path to the food. The Woods1 environment is the simplest maze, and its shortest path has, on average, a length of 1.7 steps. If the map is left, the robot re-enters on the opposite side of the map. The Maze4 environment is larger and requires, on average, 3.5 steps to reach the food. The Woods14 environment is a corner case to test the multi-step capabilities of XCS, as it requires learning a long path of up to 18 steps toward the food, with the average shortest path requiring 9.5



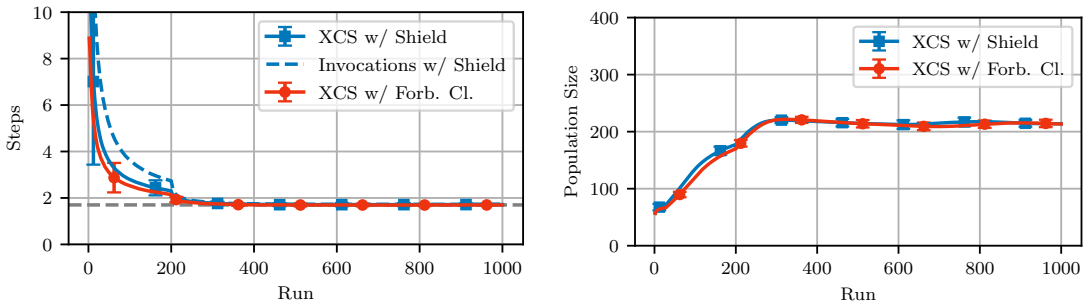
(a) Woods1 (b) Maze4 (c) Woods14

Figure 5.5: Overview of the three different maze environments used for the experimental evaluation.

steps. Therefore, a higher discount factor γ of 0.9 is employed, while the other environments use a discount factor of 0.71. The specify operator is also parameterized the same as in Chapter 4 with the threshold $n_{sp} = 20$ and specify probability $p_{sp} = 0.5$ in Woods1 and Maze4, and $p_{sp} = 0.8$ in Woods14. As it is common practice for such learning problems, a maximum number of steps per run has been defined to escape unsuccessful runs, e.g., if the robot is stuck in a loop. The run is prematurely stopped if the food has not been reached in the Woods1 and Maze4 environments after 30 steps. For the Woods14 environment, a larger step limit of 100 has been employed.

To avoid crashing into obstacles, which could damage the robot or its surroundings in a real-world environment, we have introduced eight forbidden classifiers into the population. Due to the generalized conditions of the classifiers, eight forbidden classifiers are entirely sufficient, as each classifier is responsible for preventing a crash in one direction, e.g., one forbidden classifier has a condition that matches whenever a rock is northern of the robot and then prevents the action "move north". Since the three environments have different complexities and layouts, it can be expected that the introduction of forbidden classifiers has a different impact. In the Woods1 environment, 16 empty fields exist, and for each field, XCS must determine the correct payoff prediction for the eight different actions, i.e., XCS must learn the predictions of 128 state/action pairs. Due to the forbidden classifiers, 27 pairs are already excluded from action selection, meaning that XCS' search space is reduced by approximately 21 %. In the Maze4 environment, roughly 45 % of the solution is already given through the forbidden classifiers, and in the Woods14 environment about 76 %.

For comparison, we have employed our shielded version of XCS, where the additional learning iterations that take place when the shield rejects an action do not number among the maximum step limit. This gives the shielded XCS additional learning opportunities to internalize the safety-critical knowledge embedded in the shield – at least in theory. While in the Woods1 and Maze4 environments, the additional iterations help XCS to find the shortest path more quickly, the behavior in the Woods14



(a) Steps

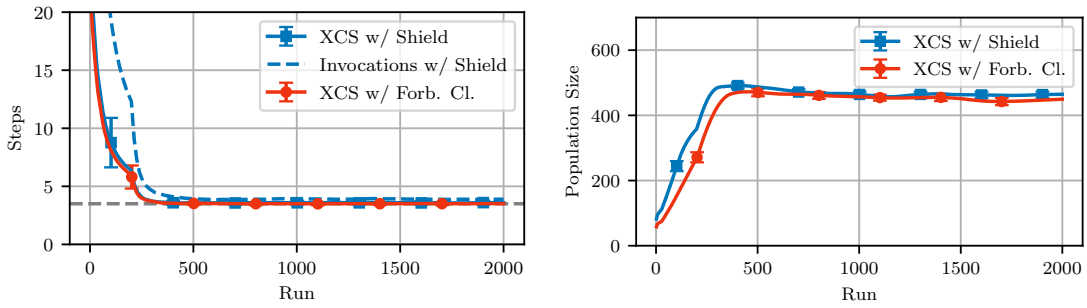
(b) Population size

Figure 5.6: Experimental results obtained in the Woods1 environment. Results are averages over 100 trials and shown as a moving average over 100 runs. The error bars visualize the observed standard deviation and the dashed line the length of the shortest path.

environment, with its larger step limit and longer paths, becomes more unstable when the interactions with the shield do not count towards the maximum step limit. This is why the limit of 100 steps in the Woods14 environment also includes the steps the shield rejected. Overall, the successful deployment of an external shield is not as straightforward as it seems and can still require some tailoring to the environment.

For the Woods1 environment, we have employed 1,000 runs; for Maze4 2,000; and for Woods14 with its longer paths 4,000 runs. For computational efficiency, it is generally favorable to employ XCS with the smallest possible population size N, because the most runtime-consuming functions of XCS operate on the whole population [13]. Hence, we have systematically determined the smallest maximum population size N for each environment at which the shielded XCS can still solve the problem. We have considered an environment fully solved if the average number of steps during the last 10 % of runs is smaller than the length of the shortest path plus a margin of 5 %. The margin is introduced because, in each run, the starting position is determined randomly, and it is thus not always possible to achieve the shortest path. We have started with a large population size N of 2,000 and then gradually decreased it by 50. For each size, we have run ten trials and determined if the environment was solved on average. As soon as this was not the case, the procedure has been stopped, and the previous size has been selected as the maximum population size for our experiments. This resulted in a maximum population size N of 300 for Woods1, 650 for Maze4, and 1,750 for Woods14.

Figure 5.6 shows the experimental results in terms of taken steps and macro-population size determined in the Woods1 environment. Until run 200, XCS with forbidden classifiers takes slightly fewer steps and requires fewer invocations of XCS due to the absence of interactions with the shield. Afterward, it seems that XCS with a shield has internalized the knowledge embedded in the shield, as it shows nearly identical behavior



(a) Steps

(b) Population size

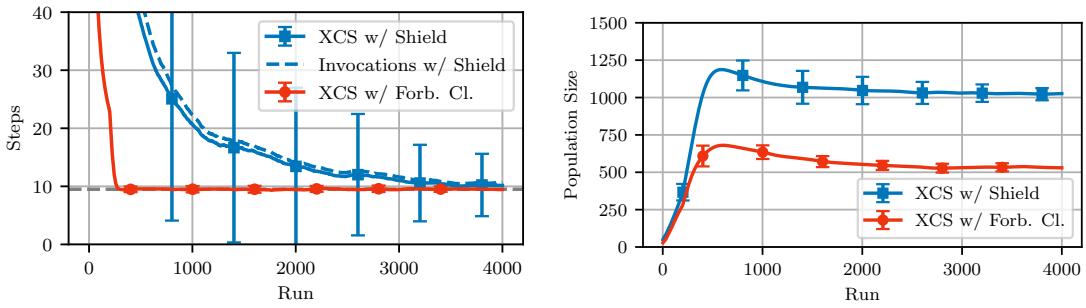
Figure 5.7: Experimental results obtained in the Maze4 environment. Results are averages over 100 trials and shown as a moving average over 100 runs. The error bars visualize the observed standard deviation and the dashed line the length of the shortest path.

to XCS with forbidden classifiers. In terms of population size, identical behavior can be observed as well. In the Maze4 environment, both versions of XCS require the same number of steps, as shown in Figure 5.7. However, the shielded version of XCS needs more invocations to achieve this. The population is initially smaller with forbidden classifiers, but after around 500 runs, the difference vanishes.

In the Woods14 environment, a more distinct advantage of forbidden classifiers can be observed, as seen in Figure 5.8. Since the number of invocations of the shielded XCS is limited to 100 in Woods14, a run can be prematurely stopped because the invocation limit is reached, but very few actual steps have been taken because the shield frequently stepped in. To denote that the food has not been reached in such runs, we have set the number of steps to the maximum of 100 once a run is prematurely stopped. This is not an issue with forbidden classifiers, as the number of actual steps and invocations is always equal. The shortest path is reliably found with forbidden classifiers after relatively few runs, as indicated by the small standard deviation. It takes considerably longer until the shortest path is found with the external shield, and the learning progress seems to be more unstable since the standard deviation is high. The classifier population with forbidden classifiers has only half the size of the population that is evolved when an external shield is used. Hence, when forbidden classifiers are used, the optimal solution is found considerably faster with a smaller population.

5.5.1 *Execution Time*

In the Woods1 and Maze4 environments, both XCS with a shield and with forbidden classifiers showed similar behavior in terms of taken steps and population size, thereby providing no clear indicator that the injection of forbidden classifiers is advantageous over an external shield. However, it has been observed that the shielded XCS is invoked more



(a) Steps

(b) Population size

Figure 5.8: Experimental results obtained in the Woods14 environment. Results are averages over 100 trials and shown as a moving average over 100 runs. The error bars visualize the observed standard deviation and the dashed line the length of the shortest path.

Table 5.1: Average CPU times of a single experimental trial.

	Woods1	Maze4	Woods14
External Shield	0.156 s	1.15 s	10.67 s
Forb. Classifiers	0.126 s	0.67 s	4.15 s
Time Saving	19.2 %	41.7 %	61.1 %

often to learn the knowledge that is embedded in the shield. This can increase the computational demand required for the execution of XCS. To quantify this, we have measured the CPU time that the execution of an experimental trial takes up on average. Table 5.1 compares the average CPU time of both employed XCS versions and shows that in the Woods1 environment, about 19 % of execution time can be saved through the use of forbidden classifiers. In comparison, the saving in the Maze4 environment is even larger with about 42 %. As seen in Figures 5.6a and 5.7a, additional invocations of XCS occur mainly at the beginning of a trial and rarely once the shielded XCS has incorporated the shield's knowledge. However, during the exploration runs, which are not included in the figures, XCS takes random actions and will thus continue to interact with the shield regardless of the current progress in finding the shortest path. The latter effect dominates the runtime savings achieved with the forbidden classifiers. Overall, the runtime savings are similar to the search space reductions of the forbidden classifiers (21 % in Woods1 and 45 % in Maze4).

The interactions with the external shield number among the maximum step limit in the Woods14 environment, which is why the effect of additional invocations of the shielded XCS is less distinct. However, since the forbidden classifiers massively reduce the search space of XCS, i.e., by 76 %, the observed runtime savings of about 61 % are still the largest among the evaluated environments. In addition to fewer invocations of

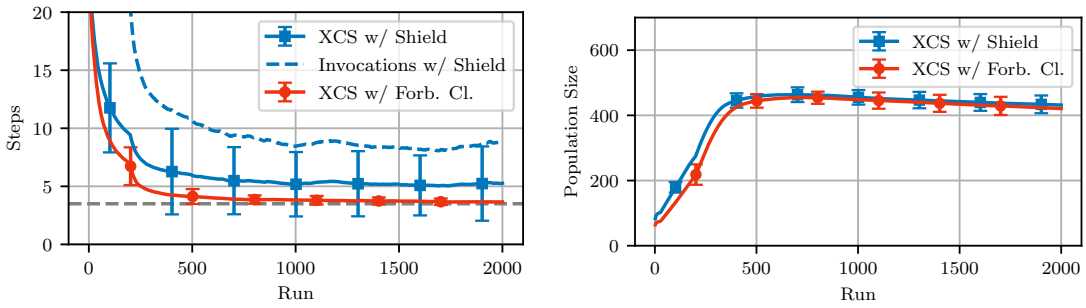
Table 5.2: Smallest maximum population sizes N at which XCS is able to solve each environment.

	Woods1	Maze4	Woods14
External Shield	300	650	1,750
Forb. Classifiers	250	400	250
Reduction	16.7 %	38.5 %	85.7 %

XCS due to the absence of the external shield, the forbidden classifiers enable XCS to find the shortest path more quickly with a smaller classifier population, which both reduces the execution time.

5.5.2 *Population Size*

The maximum population size N employed in the experiments has been determined such that the shielded XCS is just able to solve each environment fully. As shown by the previous evaluation of the 6-Multiplexer, the presence of forbidden classifiers can lead to smaller classifier populations, which has also been observed in the Woods14 environment. While the observed populations in the Woods1 and Maze4 environments have been of similar size to the shielded XCS, it is still of interest if the maximum population size N could have been decreased in case forbidden classifiers are used. To investigate this, we have repeated our methodology of determining the maximum population size N, i.e., decreasing the size until the environment is not fully solved in the last 10 % of the runs, but with XCS with forbidden classifiers instead of a shield. The results are shown in Table 5.2. The maximum population size N can be reduced in all environments when forbidden classifiers are employed. Again, the reduction of the population size depends on the reduction of the search space through the forbidden classifiers, as N can be reduced the least in Woods1 with about 17 % and by far the most in the Woods14 environment, where it can be reduced by approximately 86 %. A smaller maximum population size N not necessarily leads to a smaller computational effort required to execute XCS, as identical classifiers are summarized into a macro-classifier with a numerosity $n > 1$. Therefore, the computational effort depends on the number of macro-classifiers in the population, which is only in the worst case equal to the maximum population size N, but typically well below. However, our results show that added forbidden classifiers can aid XCS in finding the optimal solution if the configured maximum population size N is insufficient to solve the problem, which may be the case if a problem domain is more complex than expected by the system designers.



(a) Steps

(b) Population size

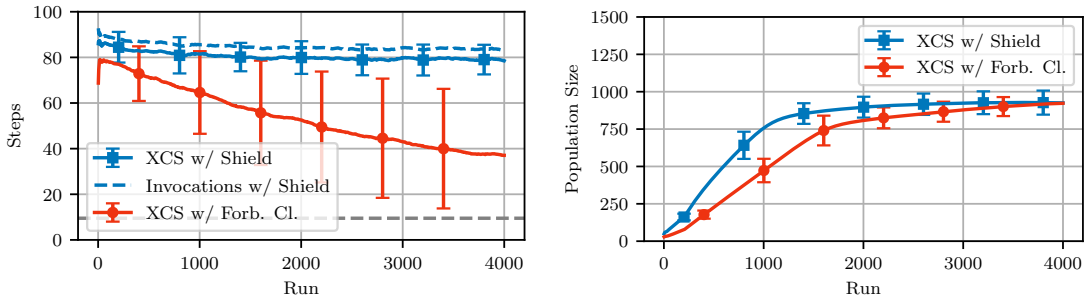
Figure 5.9: Experimental results obtained in the Maze4 environment with deactivated specify operator. Results are averages over 100 trials and shown as a moving average over 100 runs. The error bars visualize the observed standard deviation and the dashed line the length of the shortest path.

5.5.3 *Overgeneralization*

In some environments, XCS is affected by overgeneralization. This means that overgeneralized classifiers that do not always predict the correct payoff dominate the population because XCS cannot reliably determine them as inaccurate. Hence, they are not assigned a low fitness which would eventually lead to the deletion from the population [58]. For our evaluation environments, this is the case especially for Maze4 and Woods14, where the use of the specify operator has prevented overgeneralization. However, the evaluation of forbidden classifiers on the 6-Multiplexer has revealed that a classifier can sometimes safely generalize into the niches covered by forbidden classifiers. Such a classifier would normally be considered overgeneralized. However, since the forbidden classifiers prevent selecting the classifier's action for a part of its input space, it can still generate accurate payoff predictions. Therefore, it seems possible that forbidden classifiers can sanitize the adverse effects of overgeneralized classifiers. To test our hypothesis, we have repeated the experiments for the Maze4 and Woods14 environment but turned the specify operator off, which means that XCS is now affected by overgeneralization.

Figure 5.9 shows the results achieved in the Maze4 environment with a disabled specify operator. With forbidden classifiers, XCS can still find the shortest path reliably, which is no longer true with an external shield. The shortest path is, on average, not reached at the end of the experiment. Further, it requires many additional invocations of XCS through the interactions with the shield, meaning that it has not been able to learn the knowledge embedded in the shield. In terms of population size, both variants behave similarly.

The results obtained in the Woods14 environment are shown in Figure 5.8. Again, the steps taken by the shielded XCS have been set to 100 if a run is prematurely stopped because the maximum of 100 invocations



(a) Steps

(b) Population size

Figure 5.10: Experimental results obtained in the Woods14 environment with deactivated specify operator. Results are averages over 100 trials and shown as a moving average over 100 runs. The error bars visualize the observed standard deviation and the dashed line the length of the shortest path.

has been reached. The number of steps taken by the shielded XCS stays very high and close to the maximum throughout the entire experiment, which indicates close to no learning progress and demonstrates the severe effects that overgeneralization can have. On the other hand, XCS with forbidden classifiers can decrease the number of required steps more consistently, even though the improvement is accompanied by a high standard deviation and is still far from reaching the shortest path with 9.5 steps. Overall, forbidden classifiers seem to be able to prevent overgeneralization in the Maze4 environment and at least alleviate it in the Woods14 environment.

5.5.4 *Interim Summary*

Our experimental comparison with a straightforward implementation of an external shield has confirmed the observations made on the 6-Multiplexer and revealed manifold advantages of forbidden classifiers. They can considerably improve learning speed, especially in environments such as Woods14, where they notably reduce the search space. Because of fewer invocations of XCS, smaller classifier populations, and more restricted exploration, the use of forbidden classifiers also leads to shorter execution times, reducing the computational burden that the execution of XCS imposes on the computing platform. A smaller maximum population size N can be employed when using forbidden classifiers instead of an external shield, thereby allowing XCS to solve the problem with fewer classifiers or tackle more complex problems with the same number of classifiers. In addition, forbidden classifiers can partly sanitize the adverse effects of overgeneralization.

Table 5.3: Overview of the input and output attributes of the car evaluation dataset from the UCI repository.

Input	
Attribute	Values
Buying Price	vhigh, high, medium, low
Maintenance Cost	vhigh, high, medium, low
Number of Doors	2, 3, 4, 5+
Persons Capacity	2, 4, 5+
Size of Trunk	small, medium, big
Safety	low, medium, high
Output	
Car Evaluation	unacc., acceptable, good, vgood

5.6 EXPERIMENTAL EVALUATION: CLASSIFICATION

Forbidden classifiers cannot only be seen as a mechanism to inject safety guarantees into an online learning system but, more generally, as a way to insert human knowledge into the classifier population of XCS. In this section, we use the car evaluation dataset from the UCI repository [28][3] to investigate the effect that forbidden classifiers can have on the classification accuracy of XCS. Even though XCS is not explicitly designed for supervised classification tasks – unlike other learning classifier systems such as BioHEL [6] or ExSTraCS [107] – a preliminary investigation of the usefulness of forbidden classifiers in classification tasks is called for to determine if it is worthwhile to implement the concept of forbidden classifiers in LCSs tailored for classification.

The car evaluation dataset from the UCI repository represents the customer evaluation of a car based on six attributes summarized in Table 5.3, i.e., the buying price, maintenance cost, number of doors, person capacity, trunk size, and the car's safety. Each attribute has three or four possible values, meaning that each input attribute can be encoded with two bits, resulting in an overall input to XCS of 12 bits. Each sample must be categorized into one of four classes, i.e., if the car is unacceptable, acceptable, good, or very good. The dataset consists of 1728 samples and has considerable class imbalances, as the class unacceptable makes up 70 % of all samples, acceptable 22 %, and good and very good each 4 %. We have selected this dataset because its input attributes and output classes are interpretable by a broad audience, which makes it possible to derive suitable forbidden classifiers by relating the input attributes to the classes with human intuition. This contrasts with other

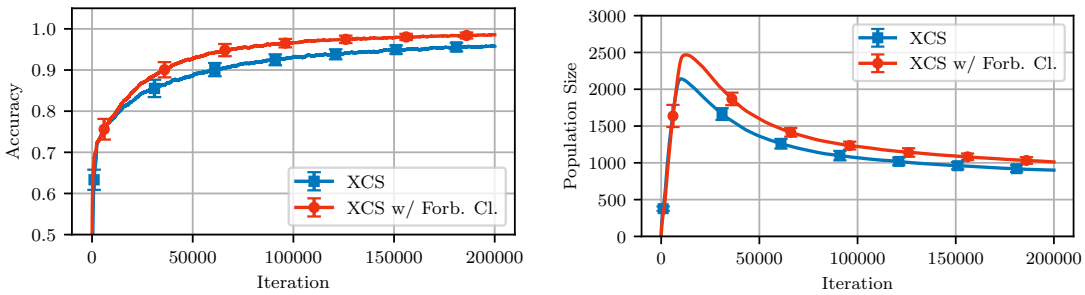
3 https://archive.ics.uci.edu/ml/datasets/Car+Evaluation, accessed 25th August 2022

popular classification datasets that require expert knowledge, e.g., from the medical domain.

Overall, we have added five forbidden classifiers. It is reasonable to assume that the safety property of a car is among the most important aspects of a car's evaluation and that an unsafe car will be seen as unacceptable. Hence, three forbidden classifiers have been added that prevent selecting the classes very good, good, and acceptable in case the car's safety is low. In addition, an evaluation as very high is prevented in case the car's safety is only medium. Since the binary encoding of the values low and medium of the input attribute safety differ by only one bit, no additional forbidden classifier was necessary. Instead, the existing forbidden classifier that prevents an evaluation of very good in case of low safety has been generalized further with an additional don't care to cover both low and medium safety. Another factor that prevents a favorable assessment by a car customer is the financial aspect, i.e., the buying price and the maintenance cost. Since a high buying price can be compensated by low maintenance cost – and vice versa – we exclude an evaluation of good and very good if the buying price and the maintenance cost are both at least high. Since the encodings of high and very high differ in only one bit, this has been achieved by adding two forbidden classifiers.

In contrast to common classification practices, we have kept the online learning nature of XCS. In each iteration, a sample from the dataset is drawn with replacement, and a reward of 1,000 is provided for correct classification and a reward of 0 in case of misclassification. While this does not allow for detecting effects like overfitting, it is more suited to the learning mechanism of XCS. In addition, this experiment aims to investigate the effect of manual knowledge injection with forbidden classifiers rather than to demonstrate the overall classification capabilities of XCS. When classifying a dataset, the expediency of the crafted forbidden classifiers can be verified by searching the dataset for samples that will inevitably be misclassified in the presence of the forbidden classifiers. This is not the case for the car evaluation dataset and the selected forbidden classifiers. However, this might be unavoidable to some extent for larger and more complex datasets, especially in the presence of noise.

In each of the 100 trials, we have employed 200,000 iterations with a maximum population size N of 6,000. As shown in Figure 5.11a, XCS achieves a classification accuracy of around 80 % rather quickly, regardless of the presence of forbidden classifiers. Considering that it is a classification problem with four classes, this can be considered as quick learning progress and indicates that XCS evolved a rough solution in relatively few iterations. Afterward, it seems that XCS is refining its population, and its classification accuracy increases only gradually. In this second phase, XCS with forbidden classifiers consistently achieves better results, with close to optimal accuracy at the end of the experiment. However, as seen in Figure 5.11b, using forbidden classifiers leads to larger classifier populations. This is a new and surprising observation, as



(a) Classification accuracy

(b) Population size

Figure 5.11: Experimental results obtained on the car evaluation dataset. Results are averages over 100 trials and shown as a moving average over 1000 samples. The error bars visualize the observed standard deviation.

in all previous experiments, the use of forbidden classifiers led to similar or smaller population sizes. An inspection of the classifier populations of XCS without forbidden classifiers revealed that several classifiers had evolved, which are similar or identical to the forbidden classifiers but have high numerosities n of more than 200. This reduces the remaining space left in the population for other classifiers. Since the inserted forbidden classifiers constantly have the minimal numerosity of 1, a larger fraction of the population can focus on the other niches when forbidden classifiers are present in the population. Apparently, the larger available space in the population is used to create more distinct classifiers, e.g., to cover specific corner samples that occur infrequently. This could also explain why the use of forbidden classifiers leads to a higher classification accuracy in the refinement phase and not in the initial phase, where the accuracy steeply increases.

Overall, our results indicate that forbidden classifiers created with domain knowledge can aid XCS when performing supervised classification of a dataset. It, therefore, seems worthwhile to investigate the concept of forbidden classifiers in other types of learning classifier systems to leverage the interpretability of their rules for manual knowledge injection.

5.7 CONCLUSION AND FUTURE WORK

We have proposed a concept to integrate safety guarantees into the learning classifier system XCS by injecting domain knowledge in the form of *forbidden classifiers*, thereby leveraging the unique interpretability of XCS' rule base. In contrast to related work, which implements safety guarantees by deploying an external shield or teacher that constantly monitors the behavior of the RL algorithm, our implementation embeds the safety guarantees directly into the classifier population of XCS. Apart from showing that their introduction requires only minor algorithmic mod-

ifications to XCS, our experimental evaluation revealed that forbidden classifiers

- open generalization opportunities not present with an external shield, overall leading to smaller classifier populations,
- relieve XCS from the computational burden of internalizing the safety-critical knowledge embedded in an external shield, thereby leading to shorter execution times, and
- can considerably improve the operating performance when XCS struggles to find the optimal solution, e.g., when it is affected by overgeneralization or the maximum population size is not large enough.

The last observation might be relevant primarily for real-world environments, where it can often be the case that XCS cannot derive a perfectly accurate payoff estimation for each state/action pair. Therefore, evaluating forbidden classifiers in practical application settings is relevant future work, along with implementing forbidden classifiers in classifier systems suited for classification. Our evaluation on a dataset from the UCI repository has shown that forbidden classifiers can be used not only to implement safety guarantees but also to inject domain knowledge and aid XCS in supervised classification tasks.

In a similar line of work, it could be promising to systematically investigate how knowledge in general could be manually added into XCS, not only by strictly preventing specific actions from being taken but instead by providing XCS with knowledge about recommended actions. For instance, the population could be bootstrapped with classifiers representing a suitable but not necessarily optimal heuristic. In contrast to the injection of forbidden classifiers, the algorithmic modifications necessary for such an approach are not intuitive, as such manually created classifiers should provide strong guidance to XCS at the beginning, but not restrict it from finding a better solution in the long term.

6

CASE STUDY: XCS FOR FREQUENCY CONTROL

So far in this thesis, XCS has been applied primarily to toy problems common in Learning Classifier System (LCS) research, i.e., multiplexer and maze environments. These environments allow for comparability to other research works and well-controllable experiments with adjustable complexity. Further, they meet all requirements necessary for XCS to evolve an optimal problem solution. This includes, for instance, that the input provided to XCS describes an individual state without state aliasing, deterministic behavior of the problem environment, and a reward function that can be precisely predicted based on the current input and chosen action. However, the use of Explore/Exploit (E/E) strategies (Chapter 4) and forbidden classifiers (Chapter 5) does not aim at deployment in such well-controllable and deterministic simulation environments. Instead, they facilitate the deployment of XCS in self-aware systems, which are used in problem environments with a high degree of uncertainty for which little apriori knowledge exists when designing the system. These environments often depict characteristics that sharply contrast those of artificial toy problem environments. Therefore, an evaluation of XCS in more practical, real-world application scenarios is called for.

This case study employs XCS for processor frequency control. Dynamic Voltage and Frequency Scaling (DVFS) is used in modern CPUs to control the computational capacity offered to running applications, the chip's power consumption, and its temperature. A system with computational self-awareness capabilities can use DVFS to balance application performance and the costs incurred by the computations while adhering to system constraints and adapting to changes in the execution environment. For example, a system could execute multiple user applications with different performance goals specified using diverse Quality of Service (QoS) metrics. At the same time, the system's power consumption should be minimized (minimization of costs), while the system's temperature must not exceed a specified threshold (system constraint). During operation, user applications leave and enter the system, requiring updates of the DVFS strategy (adaptation to changes). Traditional DVFS mechanisms, such as static heuristics and control theoretic approaches, often require extensive apriori knowledge, such as accurate system models, or cannot adapt to changes.

As an outcome, LCS variants have been successfully employed to implement autonomous and adaptive DVFS mechanisms [26, 70, 98, 117].

In this case study, we employ XCS for controlling a CPU's frequency to achieve an Instructions per Second (IPS) target of a user application while minimizing power consumption. As such, the resulting system is stimulus- and time-aware since it explicitly learns from past experiences to make more informed decisions in the future. The application scenario shares similarities with related work [26], but we apply XCS instead of a lightweight LCS with a static ruleset. In addition, we employ a CPU simulation to analyze XCS's learning behavior extensively and do not focus on a practical deployment in hardware. Hence, this case study does not primarily aim to demonstrate that XCS can evolve a superior DVFS strategy. Instead, it should confirm the observations of related work and investigate how XCS behaves in problem environments that are known to be amenable to XCS but do not depict the ideal properties of artificial toy problems. Based on the derived insights, future work can be laid out to make XCS more suitable for real-world problem environments.

Mathis Brede's C++ XCS implementation is employed in this case study, which he has extended with real-valued conditions for use in this work. In summary, the case study adds three key observations to the existing body of XCS research:

- Even though we have employed a different LCS, i.e., XCS, and executed different benchmark applications on a different CPU, we have been able to confirm the results of related work. LCSs, with their dynamic classifier generalization, outperform tabular Q-learning with fixed generalizations when performing DVFS.
- The reward that will be received is not accurately predictable, hindering XCS from successfully generalizing classifiers and preventing the use of error-based E/E strategies. The reward's predictability differs between the evaluated benchmark applications as well as between different execution phases of an application.
- To a varying degree, all benchmark applications depict an unbalanced sampling of the input state space. XCS requires a large classifier population to adequately cover infrequently occurring states, even though the number of actually occurring states is relatively small. Approaches derived from XCS theory to counteract the adverse effects of the unbalanced state sampling can only limitedly alleviate the problem.

This chapter continues by providing an overview of related work in Section 6.1. Next, the overall application scenario is presented in Section 6.2, while Section 6.3 describes the employed experimental setup. The experimental results investigating the frequency control behavior of XCS are given in Section 6.4, and Section 6.5 extensively evaluates the learning behavior of XCS. Finally, section 6.6 concludes the chapter and outlines future work.

6.1 RELATED WORK

Reinforcement Learning (RL) techniques, especially Q-learning approaches, are frequently employed for CPU management, as seen in the survey of Pagani et al. [78]. For instance, Shen et al. [90] use tabular Q-learning to optimize the energy consumption, compute performance, and chip temperature of a CPU. All three objectives are summarized into a single reward function using linear weighting. The input is the current clock frequency, temperature, IPS rate, and CPU utilization, while the action is the clock frequency applied in the next sampling period. However, tabular Q-learning enumerates all states, and the Q-table requires an entry for each possible state/action pair, which can lead to unfeasibly high memory requirements. Further, many learning cycles are needed to determine the expected payoff of each table entry reliably. To alleviate this, Iranfar et al. [48] employ a restricted form of tabular Q-learning, in which not all actions are available in all states. The objective is to reduce the thermal stress of the chip by controlling the frequency and migrating tasks. However, available for selection are only the actions that do not violate the constraints on power consumption and thermal gradient. This not only prevents violations of the constraints but also decreases the size of the Q-table.

Another alternative to tabular Q-learning is the approximation of the payoff prediction ("Q-value") with function approximation techniques. Lu et al. [68] also investigate the minimization of thermal stress. The input consists of temperature readings from several sensors distributed over the many-core CPU, while the actions allocate newly arriving tasks to cores. This results in a large state/action space, preventing the use of a tabular approach. Instead, the Q-value is approximated by a linear combination of basis functions, whose parameters are updated using a gradient descent technique. Gupta et al. [33] approximate the Q-values with a deep neural network, resulting in a deep Q-network (DQN). The input consists of continuous state variables, among them performance counters of the CPU, while the action determines the frequency and number of active cores. Optimized is the performance per watt of the executed application. On an Odroid-XU3 platform, the DQN achieved results similar to an omniscient oracle.

In LCSs, the drawbacks of large Q-tables are alleviated by employing generalized classifiers. Frequently employed for processor management are Learning Classifier Tables (LCTs) [116]. An LCT is similar to XCS but uses a pre-populated classifier population and does not evolve new classifiers at runtime. Further, it does not use the prediction accuracy as fitness metric but instead the magnitude of the payoff prediction, i.e., an LCT is a strength-based LCS. These modifications allow for resource-efficient LCT hardware implementations that depict low latencies. In [117], LCTs are deployed to control the CPU frequency and migrate tasks. The objective combines achieving a low CPU frequency, high utilization, and a homogeneous workload distribution among cores. A CPU with three

cores is evaluated in a simulation in which each core is managed by an LCT. The results show that LCTs are competitive with traditional DVFS strategies.

A similar setup is investigated in [26]. The LCTs either optimize an IPS-oriented objective function, i.e., reaching a target IPS rate while minimizing power, or a power-oriented objective function, i.e., following a reference IPS rate while adhering to a power constraint. The comparison with a control-theoretic approach yielded competitive results, even in the presence of environmental dynamics. Following a reference IPS rate through frequency control while adhering to a power constraint is also the objective pursued in [70]. Violations of the power constraint are prevented by an external supervisor that detects imminent violations and forces the system to reverse to the best configuration that has been observed so far.

GAE-LCT [98] equips LCTs with a Genetic Algorithm (GA) to evolve classifiers at runtime. GAE-LCT is more similar to XCS than the original LCT, as a classifier's fitness equals the classifier's prediction accuracy. However, in contrast to XCS, no fitness sharing is applied. The classifier conditions are interval-based, as in XCSR [97], but with discretized values. The LCT is implemented in hardware, while the GA generates new classifiers in software. During classifier generation, a validity check is performed, and classifiers are discarded if they match solely to input states that, due to physical boundaries, never occur. A local error-based E/E strategy is employed to achieve autonomy.

The input to GAE-LCT is the current CPU utilization, frequency, and IPS rate. All input values are discretized using a binning approach. To fulfill the goal of following an IPS reference while not violating a power constraint, the CPU frequency can either be increased or decreased by a unit step. Using a soft-core CPU on a Field Programmable Gate Array (FPGA), it is shown that GAE-LCT outperforms tabular Q-learning, both in terms of following the IPS reference and avoiding violations of the power budget. Further, it is shown that GAE-LCT can autonomously adapt to varying IPS references and power constraints.

Our evaluation scenario shares many similarities with GAE-LCT in that it aims at controlling the IPS rate while also considering the power consumption and evolving new classifiers with a GA. However, our evaluation revolves primarily around the LCS learning mechanism. Instead of an LCT, we apply the standard XCS(R) with a larger number of inputs and actions, thereby increasing the size of the input and action space and overall learning complexity. Further, we do not focus on an efficient hardware implementation with low latencies, as is the case for GAE-LCT, but conduct an extensive evaluation of the learning behavior of XCS.

6.2 APPLICATION SCENARIO

We employ XCS for frequency control to balance application performance and power consumption. XCS controls the CPU frequency while a user

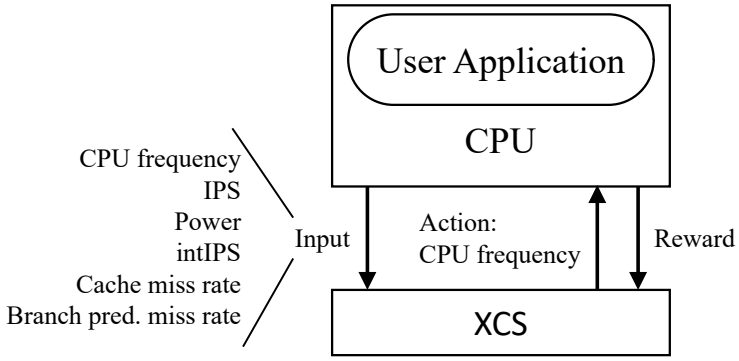


Figure 6.1: XCS controlling the CPU frequency to reach a target IPS rate.

application is executed on the CPU. The application should reach a target performance level, which is measured in terms of a QoS metric. Exceeding the desired QoS level is assumed to provide no additional benefit. Instead, XCS should minimize the power consumption of the CPU to save energy. Hence, the objective of XCS is to control the CPU frequency such that the target QoS level is reached while the power consumption is minimized.

Typically, QoS metrics are highly application-specific and diverse. For instance, the frame rate can be a relevant QoS metric for a video-processing application, while a web server could use the average response time. Using such application-specific QoS metrics for our evaluation would be somewhat cumbersome, as each user application would need to be modified to measure the relevant QoS metrics, e.g., with the heartbeat API [42]. Further, the inputs of XCS would need to be tailored to each application and its QoS metric. To facilitate a hassle-free evaluation of multiple user applications, we have used the IPS rate as a general representative for QoS metrics, as it is frequently done in related work [26, 70, 98]. The IPS rate can be measured for all applications in the same way. However, it can still show a dynamic behavior during application execution, allowing for a thorough evaluation of frequency control strategies.

The overall application scenario is depicted in Figure 6.1. XCS must be provided with sufficient and relevant input information to control the application's IPS rate and minimize the CPU's power consumption. The input encompasses the current CPU frequency, power consumption, and IPS rate. In addition, the cache miss rate is provided as an indicator of the current memory-boundness of the application, along with the branch prediction miss rate, which has been shown to be a suitable measure for the compute-boundness of an application [27]. The current rate of integer Instructions Per Second (*intIPS*) provides information about the current instruction mix, which also influences the overall IPS rate. All inputs are periodically sampled and supplied to XCS, which then decides the CPU frequency applied during the next interval. Since the CPU frequency employed during the previous interval is part of XCS's input,

the intervals can be treated independently. Therefore, XCS is used in single-step mode.

$$\text{reward} = \begin{cases} \frac{\text{IPS}}{\text{IPS}_{target}} & \text{if IPS} < \text{IPS}_{target} \\ 1 + \left(1 - \frac{\text{power}}{\text{power}_{max}}\right) & \text{else} \end{cases} \tag{6.1}$$

The reward function given in Equation 6.1 is employed to guide XCS into evolving a control strategy that is directed toward the objective of reaching the IPS target while minimizing power consumption. The reward is calculated and provided to XCS at the end of each sampling period, i.e., after the effects of the frequency set at the beginning of the period can be evaluated. As long as the IPS target is not reached, the reward increases linearly with the current IPS rate up to a maximum of 1. Once the IPS target is reached, XCS gets a constant reward of 1 for fulfilling the IPS target plus an additional fraction that depends on the current power consumption. The additional fraction takes values between 0 and 1 and linearly increases with reductions in power consumption. Hence, the overall reward function provides rewards from 0 to 2. In line with the specified objective, the reward is maximized if the IPS target is reached with minimal power consumption.

Since all inputs of XCS are real-valued, XCSR [97] with its real-valued interval conditions, as described in Section 3.3, is used. In contrast to GAE-LCT [98], the inputs are not discretized but supplied to XCS as real-valued inputs after normalization to the value range of 0 to 1. Together with the larger input and action space of XCS, this emphasizes that our evaluation is focused more on the learning characteristics of the LCS than GAE-LCT, which laid the primary focus on the hardware implementation of the LCS to achieve efficient execution and low latencies.

6.3 EXPERIMENTAL SETUP

The experimental setup used in our evaluation consists of multiple aspects. First, Subsection 6.3.1 outlines the execution platform employed to run the experiments and gather performance statistics. The benchmark applications whose execution is controlled are presented in Subsection 6.3.2, while Subsection 6.3.3 describes the parameterization of XCS, along with tabular Q-learning and a static control strategy that are employed as reference strategies. Finally, Subsection 6.3.4 details our methodology for generating training data.

6.3.1 *Execution Platform*

We conduct our experiments using the gem5 CPU simulator [11] in full-system mode, which allows for gathering all required inputs and evaluation statistics at customizable intervals. The employed CPU configuration resembles an ARM big.LITTLE architecture comprising of ARM

Cortex-A15 ("big") and Cortex-A7 ("little") cores. The instantiated CPU consists of one Cortex-A15 and one Cortex-A7 core. The user application is executed on the Cortex-A15 core, which is isolated with the `isolcpus` kernel option. This way, the core processes only the user application and basic kernel functions, and the core's simulation statistics, e.g., the current IPS rate, can be directly used to generate XCS's input. The smaller Cortex-A7 core is responsible only for executing the operating system functions. Consequently, only the Cortex-A15 core is under the control of XCS.

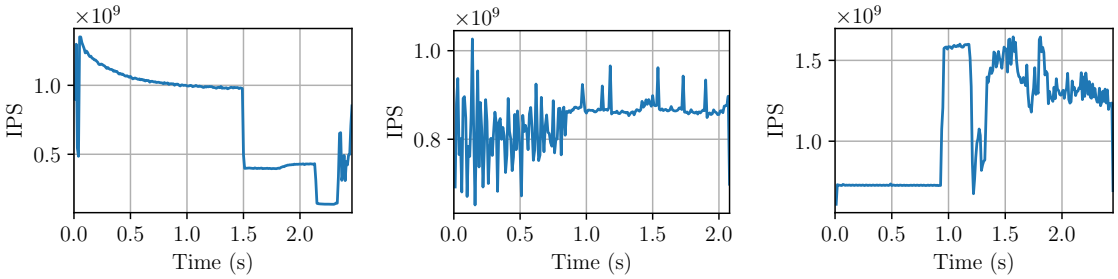
Our C++ implementation of XCS with real-valued conditions[1] is executed outside the simulation and thereby resembles a latency-free virtual hardware module. The power consumption of the core executing the user application is modeled using the power model of the ARM big.LITTLE architecture of Walker et al. [108]. It periodically samples selected CPU performance counters and calculates the average power consumption during the sampling periods based on coefficients that have been determined empirically on a physical hardware platform. The power model specifies four different CPU frequencies, i.e., 600 MHz, 1 GHz, 1.4 GHz, and 1.8 GHz. The power calculations explicitly consider the voltage that is set by the CPU's internal DVFS mechanism for each frequency. Consequently, XCS is restricted to choosing from the four specified CPU frequencies. XCS updates the CPU frequency every 10 ms, as shorter sampling periods result in noisy power values obtained from the power model.

6.3.2 *Benchmark Applications*

As user applications, we have employed benchmarks from the PARSEC benchmark suite [10], which have been cross-compiled to run on the gem5 ARM CPU using the ARM gem5 research starter kit[2]. The focus laid on benchmark applications that, according to the description given in [10], depict some form of memory-boundness during execution. It is expected that memory-bound applications result in a more diverse behavior in terms of their IPS rate, allowing for a more thorough evaluation of frequency control strategies. The applications have then been executed on the simulated ARM CPU with a static frequency of 1 GHz to characterize their runtime behavior and decide on a subset of benchmark applications suited for the experimental evaluation.

After excluding applications that either depict execution times infeasible for full-system simulations or a static behavior in terms of IPS rate, three PARSEC applications remained: Canneal, Ferret, and Freqmine. Canneal is a simulated annealing algorithm that minimizes a chip's routing costs. When executed with the `simsmall` workload of the PARSEC suite and a sampling interval of 10 ms, its IPS rate develops as shown in Figure 6.2a. The IPS rate shows a stable behavior, and different com-

1 https://git.uni-paderborn.de/xcs/xcs-real, accessed 24.03.23
2 https://github.com/arm-university/arm-gem5-rsk, accessed 10.03.2023



(a) Canneal (b) Ferret (c) Freqmine

Figure 6.2: IPS rate of PARSEC benchmark applications when executed with the `simsmall` input and a CPU frequency of 1 GHz.

putational phases of Canneal can be clearly distinguished. In contrast, Ferret, which performs a similarity search, results in an erratic IPS rate that becomes more consistent during the second phase of the application's execution, as shown in Figure 6.2b. Freqmine is a data mining application that first undergoes two phases with constant IPS rates and then transitions into a phase of varying IPS rates, as shown in Figure 6.2c. Since all three applications depict different characteristics and behavior of their IPS rates, they can test different aspects of frequency control strategies. Hence, Canneal, Ferret, and Freqmine have all been selected as benchmark applications for the experimental evaluation.

6.3.3 *Control Strategies*

XCS is compared to a static control strategy consisting of handmade decision rules and tabular Q-learning, which is an RL technique. XCS has employed a population with a maximum size N of 4,000 classifiers and a common parameterization.[3] Exceptions are the learning rate β, which has been decreased to 0.1, and the coefficient α of the accuracy function, which has been set to 0.8. Both modifications aim at making XCS more robust to noisy or unpredictable rewards. The lower learning rate decreases the impact of single rewards on the error estimate ϵ, and the higher α leads to a less drastic decrease in classifier accuracy if ϵ_0 is exceeded. Preliminary test runs have shown that XCS achieves better results with a relatively low degree of generalization, which is why the covering range s_0 and the mutation range m_0 have been set to 0.1. Since the current frequency can take only four distinct values, it is not fed into XCS as a real-valued input but as a binary input encoded with two bits. Such mixed conditions that only use real-valued inputs where necessary have already been successfully applied in the XCS implementation scikit-XCS [118].

3 That is $\beta = 0.1$, $\alpha = 0.8$, $\nu = 5$, $\mu = 0.04$, $\delta = 0.1$, $p_I = 0.5$, $\epsilon_I = 0$, $f_I = 0.01$, $P_\# = 0.3$, $\epsilon_0 = 0.02$, $\chi = 0.8$, $\theta_{GA} = 25$, $\theta_{sub} = 20$, $\theta_{del} = 20$, $s_0 = 0.1$, $m_0 = 0.1$, *DoGaSubsumption* = *True*, *DoActionSetSubsumption* = *False*.

Algorithm 1 Static frequency control strategy

```
setLowestFrequency()
changed ← True
decreased ← False
loop
    wait 10 ms
    if changed then
        IPS_ref ← IPS_current
        changed ← False
    end if

    if IPS_current < IPS_target then
        increaseFrequency()
        changed ← True
        decreased ← False
    else if IPS_current > (1 + x) * IPS_ref then
        decreaseFrequency()
        changed ← True
        decreased ← True
    else if decreased then
        decreaseFrequency()
        changed ← True
        decreased ← True
    end if
end loop
```

Tabular Q-learning is an RL technique that holds for each state/action pair the expected reward. The entries in the Q-table that contain the reward prediction p are updated similarly than in XCS, i.e., with the update equation $p = p + \beta \cdot (R - p)$, where R is the received reward and β the learning rate that has also been set to 0.1. The real-valued inputs must be discretized since Q-learning requires enumerable states. A straight-forward binning approach is applied, and Q-learning with 10, 20, and 50 bins per real-valued input has been evaluated. The expectation is that Q-learning with fewer bins initially learns more quickly, while the more fine-grained binning is superior in the long term once more learning samples have been observed.

The pseudo-code of the employed static strategy is depicted in Algorithm 1. Initially, the CPU frequency is set to the lowest possible value. If the current IPS rate is below the target IPS, the CPU frequency is increased by one step. To identify the potential for frequency reductions, the IPS rate achieved directly after a frequency change is stored as IPS_{ref}. When the IPS rate increases and an IPS rate higher than $(1 + x) \cdot IPS_{ref}$ is observed, the frequency is decreased by one step, as it could be the case that the target IPS can now be achieved with a lower frequency. If this reduction has been too eager and the IPS rate drops below the target rate,

the frequency will be increased again in the following period. Otherwise, the CPU frequency decreases further in the subsequent sampling periods until the IPS rate drops below the target rate. Thereby, the static strategy "probes" for the lowest possible CPU frequency that achieves the IPS target. The static strategy is parameterized by the parameter x, whose optimal value depends on the characteristics of the user application and cannot be determined a priori. In this experimental evaluation, x has been set to 0.05.

6.3.4 *Training Sets*

The training of XCS and Q-learning potentially requires many learning iterations, translating into multiple executions of the benchmark applications in the simulator. However, full-system simulations in the single-threaded gem5 simulator depict long execution times. Therefore, training XCS and Q-learning online by running a sequence of benchmark applications in the simulator is infeasible.

An offline learning procedure has been employed to gather statistically meaningful results in a reasonable time frame. Every benchmark application is executed multiple times in a gem5 simulation to gather training data. Every 10 ms, the relevant inputs such as IPS rate and power consumption are collected, and a random frequency is chosen for the next sampling period. Since XCS learns primarily in its exploration cycles, in which a random action, i.e., frequency, is chosen, the training sets resemble application executions that would have resulted if XCS controlled the frequency in exploration mode. The training data can then be used to train XCS in exploration mode by enforcing the selection of the action specified by the training sample instead of letting XCS select the action. This way, XCS can be trained solely with the gathered training data and without the need to perform full-system simulations. Only after the training procedure a single evaluation run in exploitation mode is conducted in the simulator to evaluate the behavior of the trained classifier population. The same training procedure is applied to Q-learning.

Each benchmark application of PARSEC has three different input workloads suited for CPU simulations. Overall, training data resulting from 20 executions with the `simsmall` workload, 15 executions with the `simmedium` workload, and 10 executions with the `simlarge` workload have been collected for each benchmark application. All collected training inputs are normalized to lie in the range between 0 and 1. In order to investigate how XCS and Q-learning behave with an increasing number of training samples, six different training sets have been defined:

(1) 3x `simsmall`, 2x `simmedium`, 1x `simlarge`

(2) 7x `simsmall`, 5x `simmedium`, 3x `simlarge`

(3) 14x `simsmall`, 10x `simmedium`, 6x `simlarge`

(4) 20x `simsmall`, 15x `simmedium`, 10x `simlarge`

(5) 2x (20x `simsmall`, 15x `simmedium`, 10x `simlarge`)

(6) 10x (20x `simsmall`, 15x `simmedium`, 10x `simlarge`)

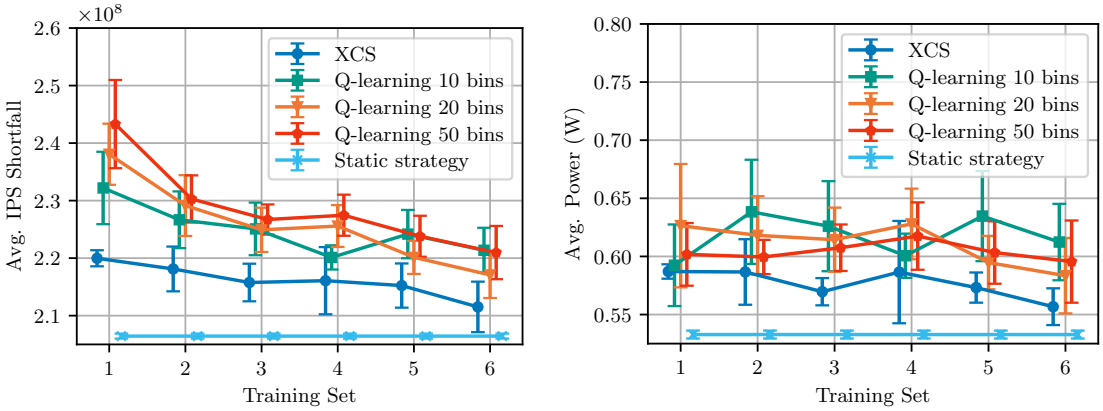
The training sets (1), (2), and (3) are created by randomly drawing subsets from the recorded executions each time a training run is conducted. Training sets (5) and (6) are compiled by using training data multiple times in randomized order. The training set (6) aims at investigating the behavior of XCS and Q-learning after many learning iterations, even though this imposes the risk of overfitting to the training data, as the same executions are part of the training set multiple times.

6.4 EXPERIMENTAL RESULTS

In order to compare and evaluate the behavior of the frequency control strategies, they are trained on the different training sets, and then their performance is evaluated in a gem5 simulation. During the simulation, both XCS and Q-learning are put into exploit mode. To evaluate the capability of managing the IPS rate, a measure must be used to quantify the extent to which the IPS rate has reached the IPS target throughout the application's execution. For this purpose, the integral of the gap between the IPS rate and the IPS target is calculated over the sampling periods where the IPS rate stayed below the IPS target. The integral is then normalized to the application's execution time to obtain a measure that we term *average IPS shortfall*. The normalization is necessary since different control strategies can lead to different execution times and thereby to more (or less) opportunities for not reaching the IPS target. Without normalization, strategies leading to IPS rates considerably higher than the target would depict shorter execution times and would have an inherent advantage. However, in our application scenario, IPS rates higher than the target are said to provide no benefit. The second evaluation metric is the average power consumption during the application's execution. According to the applied reward function, minimizing the IPS shortfall should be the control strategies' first priority, while minimizing the power consumption is the second priority.

Due to the long runtimes of full-system simulations, evaluating the control strategies on all combinations of benchmark applications and input workloads is infeasible. Hence, the evaluation of Canneal is restricted to the `simmedium` workload, while Ferret and Freqmine use the `simsmall` workload. The IPS target has been set to 1,000 Million Instructions per Second (MIPS). According to Figure 6.2, this IPS target can, in some phases, be reached easily with a CPU frequency of 1 GHz, while other phases of the applications' executions require higher frequencies. Hence, a target of 1,000 MIPS demands frequent frequency adaptations from the control strategies. For each training set and benchmark application, ten repetitions have been conducted.

The IPS shortfall and power consumption achieved when executing Canneal with the `simmedium` workload are shown in Figure 6.3a and 6.3b,



(a) IPS shortfall

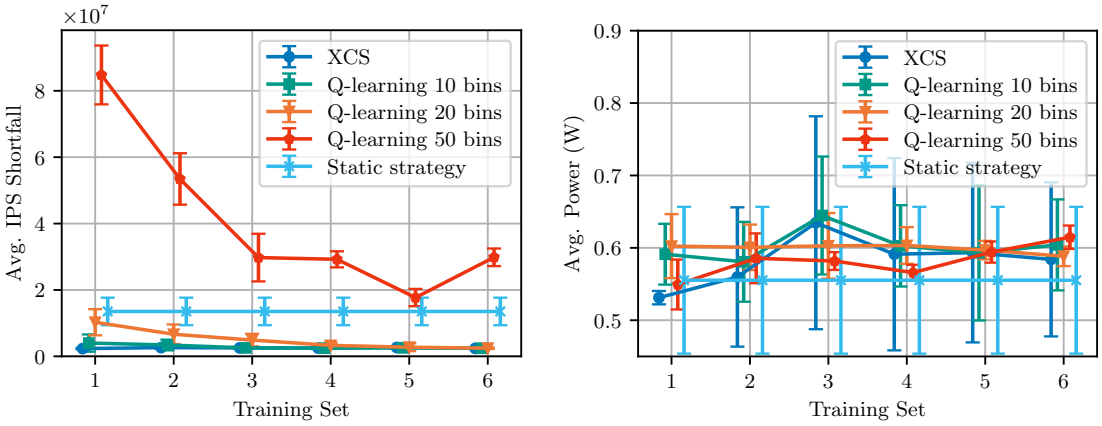
(b) Power consumption

Figure 6.3: IPS shortfall and power consumption when executing Canneal with the `simmedium` workload. Results are averages of 10 repetitions with the error bars depicting the standard deviation.

respectively. The static strategy consistently achieves the best results, both in terms of minimizing the IPS shortfall and power consumption. Q-learning shows an improvement in IPS shortfall with increasing training set size, indicating that its learning mechanism benefits from additional learning opportunities. The power consumption as the secondary objective is not improved, however. Considering that the discretization with 10 bins is relatively coarse-grained, Q-learning with 10 bins performs surprisingly well, especially on the smaller training sets. The discretization with 20 bins achieves the best results on larger training sets. Employing 50 bins seems to be a too fine-grained discretization for the given number of training samples. In terms of minimizing IPS shortfall, XCS, with its variable degree of generalization, is consistently better than all evaluated Q-learning variants. The achieved power consumption is also lower than with Q-learning, even though this observation is less distinct.

The execution characteristics of Canneal could explain the superiority of the static strategy. As depicted by Figure 6.2a, Canneal has a very steady development of its IPS rate with few but clearly distinguishable phases. This enables the static strategy to reliably determine the most suitable frequency at the beginning of each phase, which can then be kept unchanged until the next phase is entered or the IPS rate drops below the target.

When controlling the execution of Ferret on the `simsmall` workload, XCS achieves the lowest IPS shortfall as seen in Figure 6.4a. The low shortfall is achieved even when trained with the smallest training set, and larger sets lead to no improvement. Q-learning with 10 bins achieves very similar results while Q-learning with 20 bins requires larger training sets to reach the same level of performance. A discretization with 50 bins leads to results considerably worse. The static strategy does not perform well, either, considering that XCS and Q-learning with 10 and 20 bins



(a) IPS shortfall

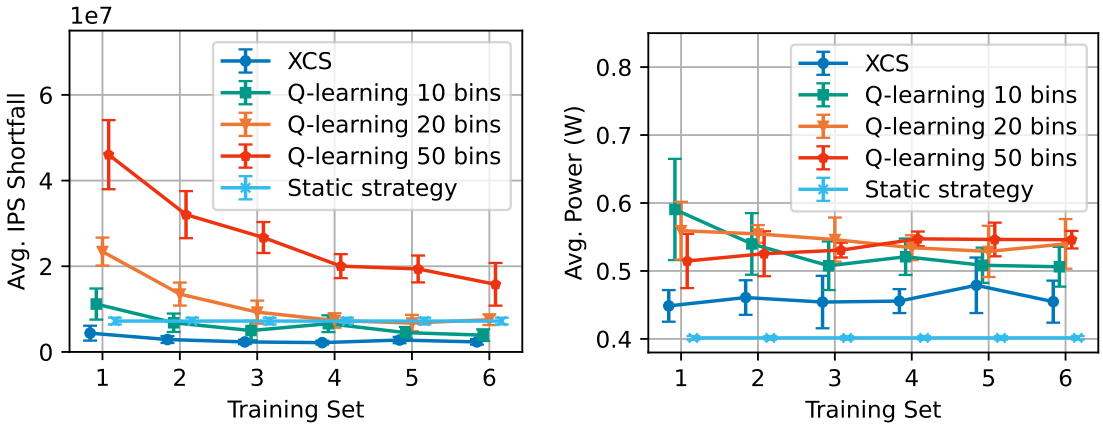
(b) Power consumption

Figure 6.4: IPS shortfall and power consumption when executing Ferret with the `simsmall` workload. Results are averages of 10 repetitions with the error bars depicting the standard deviation. XCS achieves the lowest IPS shortfall, but Q-learning with 10 bins partly hides its graph.

achieve an IPS shortfall of close to zero. Apparently, the erratic behavior of Ferret's IPS rate makes the static strategy decrease the CPU frequency unnecessarily often. This also results in the lowest power consumption, as seen in Figure 6.4b. However, due to large standard deviations, no definitive conclusions can be drawn from the power consumption measurements.

When controlling the execution of Freqmine on the `simsmall` workload, XCS again achieves the lowest IPS shortfall, as depicted in Figure 6.5a. In terms of power consumption, it achieves the second best results and is only excelled by the static strategy as seen in Figure 6.5a. It again seems to be the case that the erratic behavior of the application's IPS rate makes the static strategy decrease the frequency unnecessarily often, which leads to the lowest power consumption at the price of a considerable IPS shortfall. Q-learning works best with 10 bins and achieves an IPS shortfall between XCS and the static strategy. With more fine-grained discretizations, however, the IPS shortfall increases.

Summary. Overall, our experimental results confirm the observations from related work, i.e., GAE-LCT [98]. XCS outperforms Q-learning when controlling the CPU frequency to achieve the desired IPS rate. In contrast to GAE-LCT, our experimental evaluation used the standard XCS(R) and different benchmark applications, inputs, actions, and objectives. Hence, there seems to be a general tendency that XCS and other LCSs are superior to Q-learning when controlling the CPU frequency. The performance of Q-learning depends on the granularity of the discretization of real-valued inputs, i.e., the number of bins. However, our results indicate that the optimal granularity depends on the executed user application. This could be one reason why XCS outperforms Q-learning, as the classifier conditions in XCS are dynamically generalized during runtime by the GA.



(a) IPS shortfall

(b) Power consumption

Figure 6.5: IPS shortfall and power consumption when executing Freqmine with the `simsmall` workload. Results are averages of 10 repetitions with the error bars depicting the standard deviation.

In addition to XCS and Q-learning, a static strategy has been evaluated and achieved decent results. The static strategy can even outperform all evaluated learning approaches for user applications that depict a steady behavior in terms of IPS rate.

6.5 ENVIRONMENTAL CHARACTERISTICS AND BEHAVIOR OF XCS

The experimental results show that XCS can adequately control the CPU frequency to fulfill a non-trivial objective. However, such real-world application domains can depict different characteristics than the toy problem environments used in Chapters 4 and 5 and in the majority of LCS research literature. Therefore, a closer investigation of the learning behavior of XCS can reveal different characteristics of the problem environments, how they impact the learning mechanism of XCS, and what aspects must be considered when employing XCS in real-world application domains.

Even though the evaluation in Section 6.4 did not investigate the internal learning behavior of XCS and focused only on statistics relevant to the application domain, it still provides initial indicators that XCS behaves differently than in toy problem environments. For instance, the preliminary parameter optimization based on a few test runs showed that small covering and mutation ranges s_0 and m_0 yield the best results. Typically, XCS requires larger ranges to first cover the whole input state space and then refine the generalizations with its GA.

When controlling the execution of Freqmine and Ferret, the performance of XCS does not improve considerably with additional training samples, as the best results are already achieved with the smallest training set. On the other hand, when controlling the execution of Canneal, XCS improves with larger training sets. However, even when trained

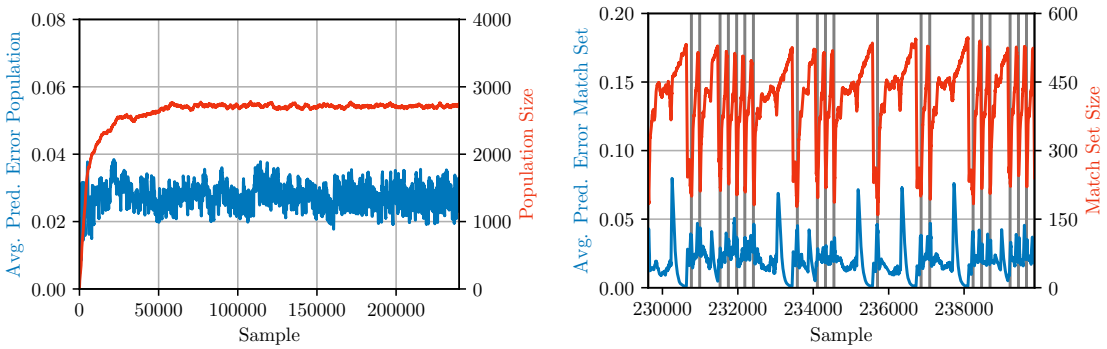
with the largest training set, it cannot reach the performance of the static strategy. Overall, this indicates that the controlled user application has a considerable impact on XCS's behavior and speed of learning.

The investigation of XCS's learning behavior is structured into three steps: First, the learning behavior of XCS is investigated in Subsection 6.5.1 by evaluating a selection of XCS's internal statistics during the training process. The results indicate that the benchmark applications have different characteristics, and some do not align well with XCS's learning mechanism. However, this deduction is inconclusive, as it can result from both the benchmark applications and XCS's dynamic generalization. To this end, Subsection 6.5.2 investigates the training procedure with Q-learning. With its fixed generalization, it allows for pinpointing the different characteristics of the benchmark applications. With the insights about these characteristics, an attempt toward tailoring XCS's hyperparameters to the application domain is made in Subsection 6.5.3. Finally, Subsection 6.5.4 discusses the implications of the gathered results.

6.5.1 *Learning Behavior of XCS*

In order to study how XCS learns to control the CPU frequency, the training set 6 has been employed, i.e., the largest training set in which each recorded execution of the benchmark applications occurs ten times. Throughout the training, relevant internal statistics of XCS are collected. The statistics include the size of the classifier population, which describes the generalization behavior, and the average prediction error of all classifiers in the population, representing the general prediction accuracy of the population. In addition to population statistics, the properties of the match set have also been investigated to reveal differences between environmental niches. Specifically, the average prediction error of the classifiers in the match set has been collected to investigate the prediction accuracy across different niches and the size of the match sets to observe the allocation of classifiers over the input state space. When determining the match set size, classifiers are counted with their numerosity n. The match set statistics have been evaluated only for the end of the training process, i.e., for a trained classifier population, to focus on underlying patterns and exclude inferences of the stochastic learning process at the beginning of training. The order in which the application executions occur in the training set is fixed over all experiments. Only the result of a single training run is shown, as averaging over multiple runs can hide relevant observations. However, the general trend of the results has been confirmed by performing additional runs.

Figure 6.6a shows the population size and the average prediction error of the population over the course of the training process for Canneal. The population size does not decrease throughout training. It is also not considerably below the maximum size of 4,000, indicating that few classifiers in the population have a numerosity n of more than one and that the classifiers depict a low degree of generalization. Otherwise,



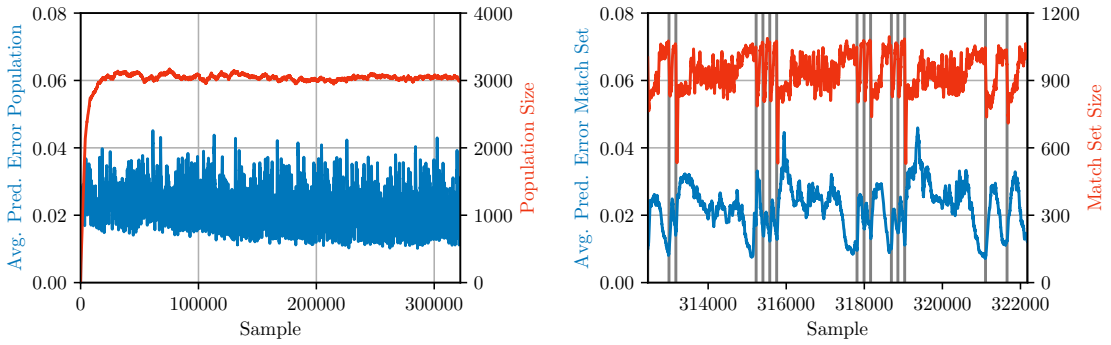
(a) Population

(b) Match set

Figure 6.6: Population and match set statistics of XCS when trained with the Canneal training set. Results are visualized as a moving average over 30 samples. The match set statistics are shown only for the end of the training process. The vertical grey lines denote the start of a new execution of Canneal.

the population size would have decreased with an increasing number of training samples, as classifier subsumption uses general classifiers to delete more specific classifiers. The prediction error oscillates and is above the error threshold ϵ_0 of 0.02. Although the visualized moving average is computed over 30 samples, which is a relatively small window compared to the number of overall samples, the degree of oscillation is relatively high. Depicted is the average error of the classifier population, but only a subset of classifiers is updated in each iteration, which should inherently constrain oscillation. In addition, the prediction error is initially not visibly higher than at the end of training. Typically, the error is high at the beginning of training and decreases as XCS learns proper generalizations. The observed behavior could result from the small covering and mutation ranges s_0 and m_0, which initially lead to the creation of specialized classifiers with a low degree of generalization but accurate predictions.

The average prediction error of the match sets is shown in Figure 6.6b. It differs considerably throughout the execution of Canneal. Most distinctively is a surge in prediction error approximately in the middle of the long-running executions with the `simlarge` workload. The moving average even hides the surge's true magnitude. The surge in prediction error corresponds to the sudden drop in IPS rate, as seen in Figure 6.2a, which decreases the reward. The sudden decrease in the received reward seems to be unpredictable for XCS. After the prediction error has increased, it gradually approaches zero, which means that either more accurate classifiers are beginning to match or that the prediction of already matching classifiers decreases. Since it is a steady and repeatedly observed pattern, it is likely that the same classifiers match but that their prediction error decreases. This can be the case when the same input state and reward are repeatedly observed, which drives the prediction error of matching classifiers to zero, regardless of whether they are adequately generalized



(a) Population

(b) Match set

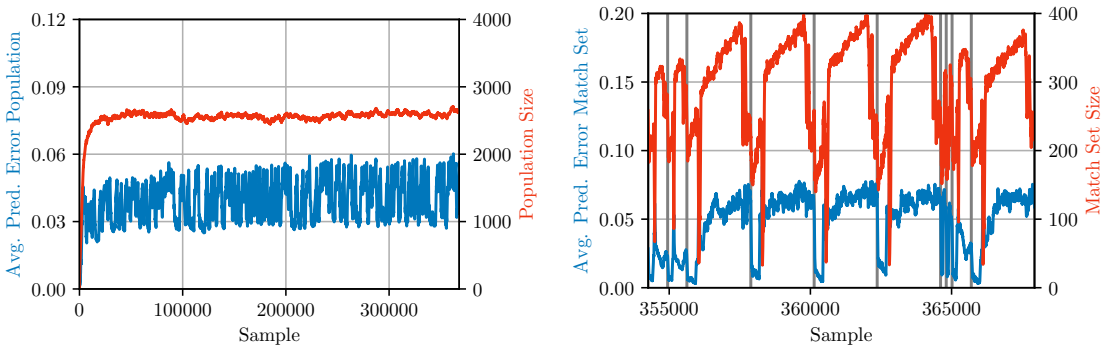
Figure 6.7: Population and match set statistics of XCS when trained with the Ferret training set. Results are visualized as a moving average over 30 samples. The match set statistics are shown only for the end of the training process. The vertical grey lines denote the start of a new execution of Ferret.

or not. Since Canneal enters a phase with steady behavior after the IPS rate has dropped, this likely explains the observed behavior.

The size of the match set is small at the beginning of an execution of Canneal but increases afterward. After the drop in IPS rate, the size of the match sets increases continuously, which can result from more classifiers beginning to match or the GA generating additional classifiers, which match the current input state. The different match sets have sizes considerably different, indicating an uneven allocation of classifiers to environmental niches. This can result from an unbalanced sampling of the input state space, which poses a reasonable explanation since repeated observations of the same input state can explain the behavior observed for the match set's prediction error.

When trained on the Ferret dataset, the average prediction error of the population oscillates even more than with Canneal, as shown by Figure 6.7a. The population size decreases neither and is even slightly larger than with Canneal. Figure 6.7b shows the size of the match sets, which do not vary as much as for the execution of Canneal. However, the match sets are roughly twice as large, which means that the population's classifiers are more generalized or that the executions of Ferret result in fewer distinct input states that are observed, allowing XCS to hold more classifiers per state. Similarly, the prediction error of the match set is also more consistent than for Canneal, even though it is still higher than the error threshold ϵ_0 of 0.02 most of the time. This could explain the large classifier population, as it prevents classifier subsumption. Overall, the unbalanced sampling of the input state space seems less distinct than for Canneal, but the oscillating prediction error still prevents successful generalization.

As seen in Figure 6.8a, training on the Freqmine dataset also results in an oscillating prediction error of the population. The prediction error is

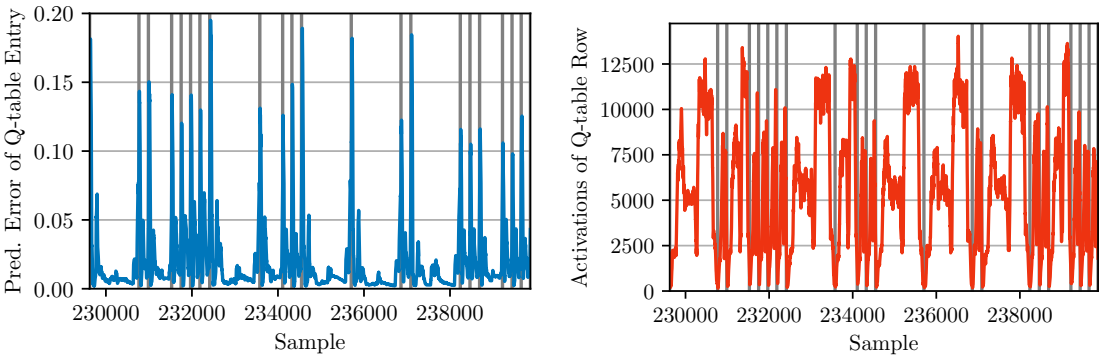


(a) Population (b) Match set

Figure 6.8: Population and match set statistics of XCS when trained with the Freqmine training set. Results are visualized as a moving average over 30 samples. The match set statistics are shown only for the end of the training process. The vertical grey lines denote the start of a new execution of Freqmine.

even higher than for the other two benchmark applications. The population size does not decrease and shows no sign of successful generalization. The size of the match sets is shown in Figure 6.8b and depicts an irregular pattern. At the beginning of Freqmine's execution, the match sets are small, indicating that the classifier population sparsely covers them. In contrast, the following execution phases are extensively covered by large match sets. Even though the input states of these phases appear to occur rather often, the prediction error of the corresponding match sets remains relatively high, with a value of around 0.05, indicating that the reward cannot be accurately predicted in these phases. In addition, the order in which Freqmine's executions occur in the training set makes a visible difference. The majority of Figure 6.8b consists of a sequence of four long-running executions with the `simlarge` workload, and an increase in the size of the match sets can be observed across the different executions. The sequence of long-running executions is then interrupted by three shorter-running executions with the `simsmall` and `simmedium` workloads. The last execution with the `simlarge` workload then depicts match sets considerably smaller than in the previous long-running executions. It could be the case that classifiers can only be reused limitedly across different workloads of Freqmine and that classifiers created during runs with the `simsmall` or `simmedium` workload do not match in runs with the `simlarge` workload.

Interim Summary. On all three benchmark applications, XCS behaves differently than on the toy problems commonly used in LCS research. In no case proper generalization in the form of a decrease in population size has been observed. The experimental results indicate that this can be due to an unbalanced sampling of the input state space, reward variance preventing accurate predictions, or, as a consequence of both effects, an inadequate parameterization of XCS. However, observing the match set



(a) Prediction error

(b) Activations per row

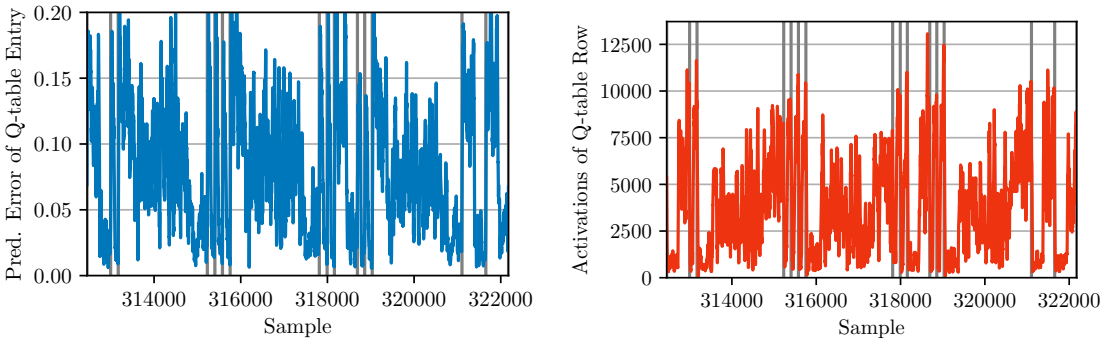
Figure 6.9: Prediction error and the number of activations of the Q-table with 50 bins when trained with the Canneal training set. Results are visualized as a moving average over 30 samples. Shown is only the end of the training process. The vertical grey lines denote the start of a new execution of Canneal.

size and prediction error of XCS does not allow for drawing definitive conclusions on the characteristics of the problem environment. Therefore, the following subsection studies the characteristics of the benchmark applications by applying tabular Q-learning and eliminating the effect of dynamic generalization.

6.5.2 *Application Characteristics*

When using XCS, the dynamic evolution of generalized classifiers prevents precisely pinpointing the causes for the observed irregular learning behavior. For instance, an oscillating prediction error can result from both reward variance that makes the reward unpredictable and inadequate classifier generalization. To isolate the effects of the problem environment, i.e., of the different benchmark applications, Q-learning instead of XCS has been applied in the training process. With the binning of the real-valued inputs, Q-learning applies fixed generalizations, which allows for studying the properties of the benchmark applications without the interference of dynamic classifier generalization. Employed has been Q-learning with 50 bins, as it applies the least amount of generalization and allows for the most fine-grained evaluation of application characteristics.

In analogy to the evaluation presented in the previous subsection, the prediction error and the sampling of the input state space have been investigated. For this, each entry of the Q-table has been extended with the estimated prediction error that is updated in the same way as in XCS. In addition, to investigate the frequency with which the different input states occur, each row of the Q-table, which includes the entries of all actions for a given input state, has been equipped with a counter that



(a) Prediction error

(b) Activations per row

Figure 6.10: Prediction error and the number of activations of the Q-table with 50 bins when trained with the Ferret training set. Results are visualized as a moving average over 30 samples. Shown is only the end of the training process. The vertical grey lines denote the start of a new execution of Ferret.

holds how often the row was activated because the corresponding input state occurred.

When the Q-table with 50 bins is trained on the Canneal dataset, the achieved prediction errors, shown in Figure 6.9a, are lower than the errors achieved with XCS. Hence, the prediction accuracy of XCS seems to be negatively affected by its classifier generalization. However, especially at the beginning of an execution of Canneal, the prediction error is relatively high, indicating that the reward is largely unpredictable in these states. A straightforward way to tackle reward variance is quantizing the reward, e.g., by rounding the received reward to discrete levels. We have applied different levels of rounding precision, i.e., 0.02, 0.05, and 0.1, but observed no visible difference in the prediction errors. This indicates that the reward's unpredictability is likely not related to a small amount of random noise but inherent to the application. The cumulated number of activations per row is shown in Figure 6.9b and confirms that the input state space suffers from unbalanced sampling, as some states occur more frequently than others.

The experimental results obtained with Q-learning and the Ferret dataset are shown in Figure 6.10. The matching entries depict a high prediction error, and rounding the reward did not reduce the prediction error, either. Again, state imbalances can be observed, with the states corresponding to the beginning of an execution occurring less frequently than other states.

A noteworthy observation can be made if the experiment is conducted with Q-learning and 10 instead of 50 bins. As shown in Figure 6.11, the prediction error of the matching entries is considerably lower, which could explain why Q-learning with 50 bins did not yield good results in controlling the IPS rate (cf. Figure 6.4a). However, this observation seems to be counterintuitive. With fewer bins, Q-learning summarizes

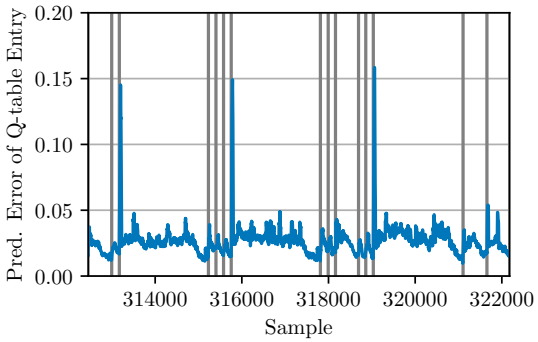
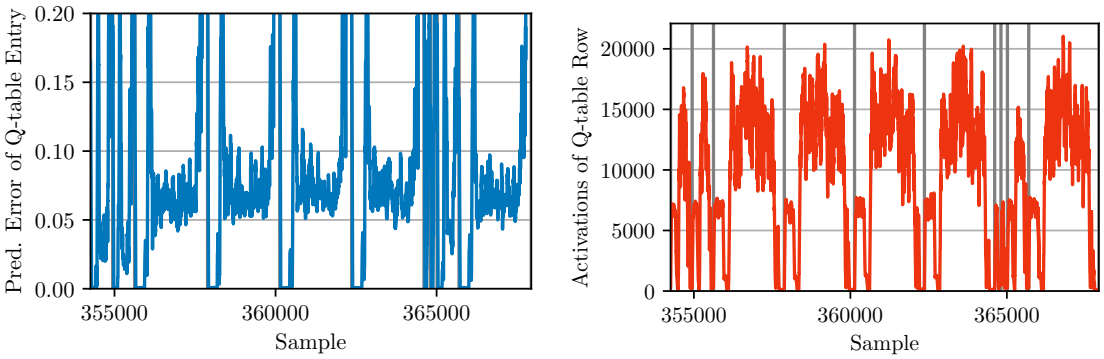


Figure 6.11: Prediction error of the Q-table with 10 bins when trained on the Ferret training set. Results are visualized as a moving average over 30 samples. Shown is only the end of the training process. The vertical grey lines denote the start of a new execution of Ferret.

more input states with different rewards in a single entry, which should increase the prediction error of the entries instead of decreasing it. A possible explanation could be that the different executions of Ferret depict different characteristics, such that the same states lead to different rewards across the different executions. With fewer bins, the entries of the Q-table are updated more frequently during a single execution, allowing it to adapt its prediction to the individual execution gradually. Even though the exact reasons for this phenomenon remain unclear, it shows that XCS, with its dynamic classifier generalization, can be superior to approaches with fixed generalization, as it can adjust the degree of generalization to minimize the prediction error.

As shown in Figure 6.12a, the Freqmine training set leads to a high prediction error. At the beginning of an execution, the error is close to zero, followed by a short spike with a very high prediction error. Most of the time, the error is above 0.05 and thus in a similar range than with XCS. Again, rounding the received rewards did not result in a visible improvement. Figure 6.12b shows that the input state space is sampled unevenly. Especially at the beginning and the end of an execution, some states are observed very infrequently, which could explain the high prediction errors in these states.

Interim Summary. With tabular Q-learning, it has been confirmed that all three benchmark applications lead to an unbalanced sampling of input states, even though the extent differs between the applications. When executing Canneal, the reward can be predicted with relatively high accuracy in most states. In contrast, the prediction accuracy during the execution of Ferret depends heavily on the granularity of the generalization. Even though it is counter-intuitive that a discretization with more bins leads to higher prediction errors, this can explain why XCS, with its evolved generalizations, outperforms Q-learning. On Freqmine, the prediction error is also relatively high but differs considerably between different states. Rounding the received rewards has been investigated as a straightforward way to make the reward more predictable and decrease



(a) Prediction error

(b) Activations per row

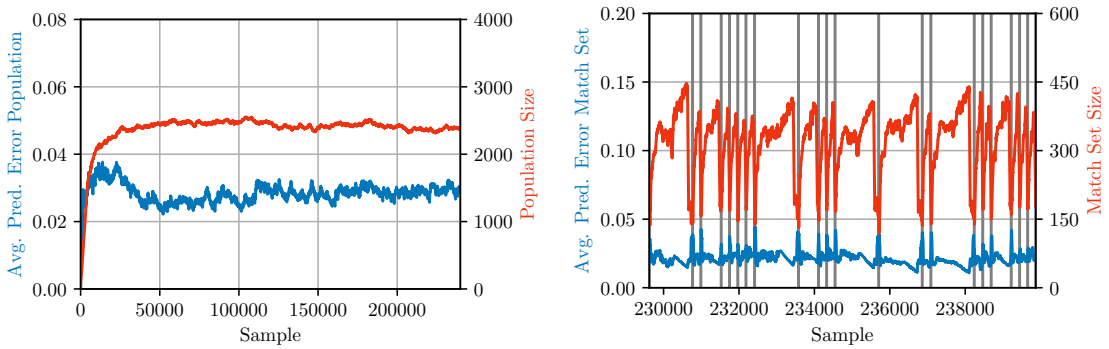
Figure 6.12: Prediction error and the number of activations of the Q-table with 50 bins when trained with the Freqmine training set. Results are visualized as a moving average over 30 samples. Shown is only the end of the training process. The vertical grey lines denote the start of a new execution of Freqmine.

the prediction errors, but with no success. This could indicate that the unpredictability of the reward is not related to small amounts of random noise but that the reward is inherently unpredictable, e.g., because the inputs provided to Q-learning and XCS at the beginning of a sampling period are insufficient to predict the state of the application execution at the end of the sampling period when the reward is determined.

6.5.3 *Hyperparameter Tailoring of XCS*

With the gained insights into the problem environment, an attempt toward tailoring the hyperparameters of XCS has been made. With optimized hyperparameters, XCS might counteract some of the irregularities observed during training. Tackled first is the unbalanced sampling of input states, which has already been investigated for XCS by Orriols-Puig et al. [77]. The primary risk in such environments is that classifiers matching to frequently occurring input states receive more reproductive opportunities in the GA, making them prone to overgeneralize into infrequently occurring niches. The theoretical analysis presented in [77] analytically derives hyperparameter values. However, this requires detailed information on the problem environment that cannot be determined for most real-world environments, such as the imbalance ratio or the number of environmental niches. Nevertheless, the derived guidelines for setting XCS's hyperparameters can still be applied to alleviate the effects of the unbalanced state sampling. They suggest using

- a small learning rate β,
- a low GA frequency, i.e., high θ_{GA},
- and a sharp distinction between fit and unfit classifiers.



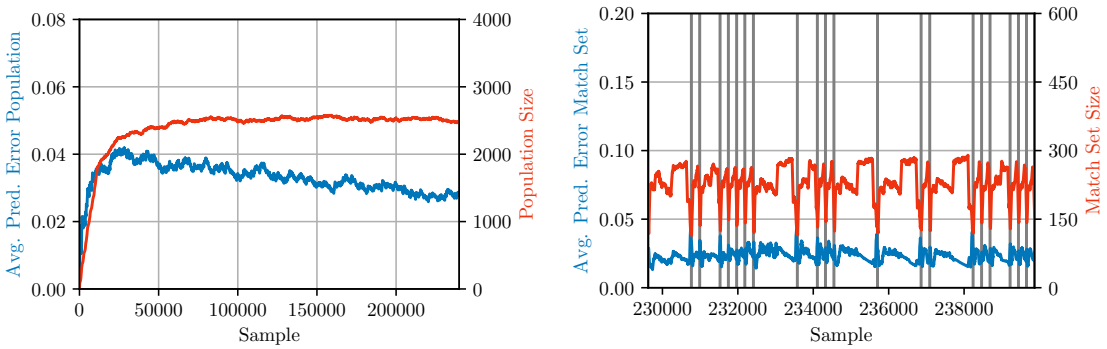
(a) Population

(b) Match set

Figure 6.13: Population and match set statistics of XCS with $\beta = 0.01$ when trained with the Canneal training set. Results are visualized as a moving average over 30 samples. The match set statistics are shown only for the end of the training process. The vertical grey lines denote the start of a new execution of Canneal.

A small learning rate β leads to lower oscillation in classifier parameters and allows for identifying overgeneral classifiers more reliably. On the downside, it can reduce learning speed and adaptivity in the face of environmental dynamics. Increasing θ_{GA} reduces the overall number of genetic events but distributes them more evenly across the environmental niches if they occur unbalancedly. Since the GA is activated less frequently, higher values of θ_{GA} can also slow down the learning process. Similar to a reduced learning rate, the sharp distinction between fit and unfit classifiers aims to identify overgeneral classifiers more reliably. It can be achieved by modifying the accuracy function via ν and α to assign classifiers a low accuracy once the prediction error ϵ is above the error threshold ϵ_0. However, all evaluated benchmark applications have depicted rewards that are not entirely predictable, leading to oscillating prediction errors. This oscillation increases the risk that an adequately generalized and accurate classifier is occasionally considered inaccurate, i.e., unfit, which means its likelihood of deletion increases if the accuracy function sharply distinguishes between accurate and inaccurate classifiers. Therefore, the parameter tailoring aiming at the unbalanced input state sampling is restricted to the learning rate β and the GA threshold θ_{GA}.

As the first step, the learning rate β has been reduced from 0.1 to 0.01. As seen in Figure 6.13a, the size of the population is reduced from approximately 3,000 to now 2,500 classifiers when trained for Canneal. However, no tendency for generalization and a corresponding decrease in population size can still be observed. Due to the reduced learning rate, the prediction error of the population oscillates less, and it is observable that the prediction error is higher at the beginning of training than at the end. The classifiers are still unevenly allocated to the environmental niches, as seen in Figure 6.13b. However, the largest match sets that have



(a) Population

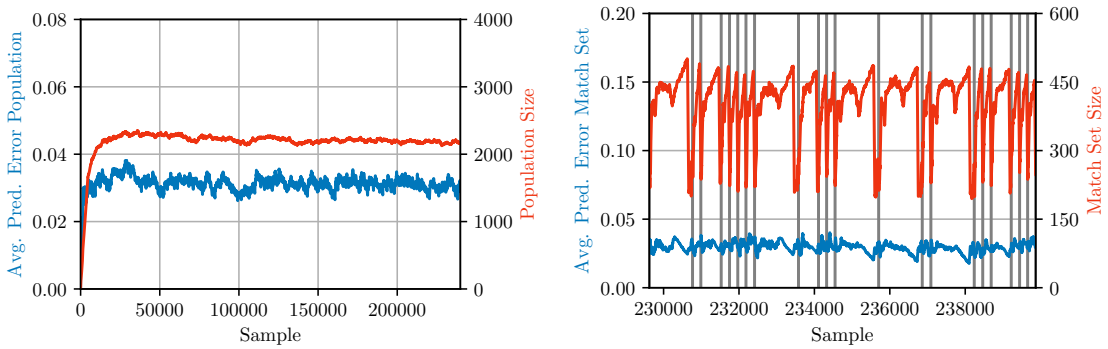
(b) Match set

Figure 6.14: Population and match set statistics of XCS with $\beta = 0.01$ and $\theta_{GA} = 200$ when trained with the Canneal training set. Results are visualized as a moving average over 30 samples. The match set statistics are shown only for the end of the training process. The vertical grey lines denote the start of a new execution of Canneal.

been observed with the initial learning rate of 0.1 contained more than 500 classifiers, while the maximum size is reduced to slightly above 400 with the lower learning rate. This indicates at least mild positive effects of the reduced learning rate toward an even classifier allocation in the population. The curve depicting the prediction error of the match sets is smoothed due to the smaller parameter updates, but the primary characteristics of the execution of Canneal are still visible. At the beginning of an execution, a small spike can be observed, indicating an unpredictable reward in this phase of execution. During long-running executions with the `simlarge` workload, a phase of continuously decreasing prediction errors is observable, even though it does not reach an error of zero due to the smaller parameter updates. The results obtained on the training sets of Ferret and Freqmine are not shown, as they depict no visible difference to the results obtained with $\beta = 0.1$ except for the smoothing of error curves.

In addition to the reduced learning rate, the GA threshold θ_{GA} has been increased from 25 to 200 to achieve a lower GA activation frequency. When trained on the Canneal training set, the population size remains unchanged, as seen in Figure 6.14a. However, the prediction error of the population decreases continuously but, in the end, reaches a value similar to the original GA threshold θ_{GA} of 25. Therefore, it seems that a more frequently activated GA has a beneficial impact on prediction accuracy.

The prediction error of the match sets at the end of the training process is unaffected by the reduced GA activation frequency, as seen in Figure 6.14b. However, the size of the match sets is reduced considerably. Considering that the overall size of the classifier population remained unchanged, this implies that the population's classifiers occur in fewer match sets, i.e., they are less generalized. Overall, a more frequent application of the GA seems to improve the average prediction accuracy



(a) Population

(b) Match set

Figure 6.15: Population and match set statistics of XCS with $\beta = 0.01$ and $\epsilon_0 = 0.05$ when trained with the Canneal training set. Results are visualized as a moving average over 30 samples. The match set statistics are shown only for the end of the training process. The vertical grey lines denote the start of a new execution of Canneal.

of the population and leads to more generalized classifiers, which is precisely the intended behavior of the GA. This contrasts with the other experimental results discussed so far, as they indicated that the GA plays a relatively unimportant role due to a lack of generalization behavior. The GA, in principle, seems to function as intended, but it hits a boundary preventing it from reducing the overall population size. The same applies to reducing the prediction error, which still has the potential to be reduced further, as can be seen in the evaluation with Q-learning (cf. Figure 6.9a). The reasons for this behavior are unclear. One possibility could be that only a small fraction of the input state space is sampled and that the classifiers are generalized into the never-occurring state space. This generalization does not impact the classifier's prediction accuracy but can hamper the use of classifier subsumption to reduce the population size.

On both the Ferret and Freqmine training sets, a decrease in the match set size can also be observed. Still, the size of the classifier population remains unchanged in both cases. Further, neither the prediction error of the population nor of the match set shows a visible difference to the case with a GA threshold θ_{GA} of 25.

The reduction of the learning rate β showed mild improvements toward an even classifier allocation of XCS by partly counteracting the adverse effects of the unbalanced input state sampling. In contrast, lowering the GA activation frequency did not yield positive outcomes and slowed the learning process. Hence, the parameter optimization has been continued with the reduced learning rate β of 0.01 and the initial GA activation threshold θ_{GA} of 25.

The last aspect considered in optimizing XCS's parameters is the non-zero prediction error due to reward variance. The prediction accuracy is the guiding criterion for the learning mechanism of XCS. Hence, a

reward function that is unpredictable to a certain extent might not be able to provide sufficient guidance to the GA. Quantizing the received reward through rounding with different granularities did not improve the prediction accuracy of Q-learning, as seen in Subsection 6.5.2. Hence, it is not expected to provide any benefits to the prediction accuracy of XCS, either. Instead, we investigate an increase of the error threshold ϵ_0 of 0.02 to 0.05. Since ϵ_0 determines up to which prediction error a classifier is considered fully accurate, this opens more generalization opportunities and, consequently, the potential for smaller classifier populations. On the downside, higher prediction errors are considered acceptable, which can lead to the more frequent selection of non-optimal actions and a decrease in operational performance, e.g., in terms of IPS shortfall. However, if XCS cannot accurately cover the entire input state space with the initial error threshold, a higher threshold can also lead to decreased prediction errors. Whether the benefits of a higher error threshold exceed the adverse side effects depends on the problem environment. In our case, positive effects can be expected especially for the benchmark application Freqmine which depicts prediction errors considerably higher than $\epsilon_0 = 0.02$ (cf. Figure 6.8b).

When trained on the Canneal training set, a very weak tendency to decrease the size of the classifier population can be seen in Figure 6.15a. However, the error threshold of 0.05 also leads to an increase in the overall prediction error, indicating that the additional generalization opportunities are exploited at the cost of prediction accuracy. This is also observed in Figure 6.15b, where the larger value for ϵ_0 results in larger match sets, indicating more generalization, and higher prediction errors in the match sets. Increased prediction errors and larger match sets have also been observed on the Ferret training set (not shown) but not with an accompanied decrease in population size. The same observations have been made for the Freqmine training set, indicating that no apparent benefits result from a higher error threshold. This is contrary to our expectation that Freqmine would benefit the most due to its inability to reach prediction errors close to the initial ϵ_0 of 0.02.

Interim Summary. Although a prior analysis of the characteristics of the benchmark applications has been conducted, optimizing the hyper-parameters of XCS to the application domain seems challenging. A lower learning rate and GA activation frequency have been employed to tackle the undesired side effects associated with the unbalanced sampling of the input state space. Both approaches have been developed based on XCS theory and have proven successful on toy problems [77]. In our case, the reduced learning rate showed mild improvements toward a more even allocation of classifiers to environmental niches. In contrast, the reduced GA activation frequency only slowed down the learning process of XCS. The reward variance present for all benchmark applications can prevent XCS from deriving adequate generalizations, and rounding the received reward to discrete levels does not alleviate the variance. Instead, we have increased the error threshold ϵ_0. According to the internal mechanics of

XCS, this should trade prediction accuracy for generalization, leading to smaller classifier populations. However, while we have observed larger prediction errors for all benchmark applications, a mild reduction in population size has been observed for only one application. Overall, this shows that tailoring XCS's hyperparameters toward the characteristics of a real-world application domain that does not depict the same characteristics as toy problems commonly used in research is quite challenging and, if possible, likely requires an exhaustive analysis of the problem environment.

6.5.4 *Implications*

Section 6.4 demonstrated that XCS performs well when controlling the CPU frequency during the execution of the three selected benchmark applications. It outperformed Q-learning and was superior to the static strategy in two out of three cases. Naturally, the question can be raised why the observed irregularities of XCS's learning behavior are of any importance if the operational performance in the application domain, i.e., frequency control, is sufficiently good. And indeed, the implications derived from the presented investigations of XCS's learning behavior do not primarily concern the specific investigated application scenario but other potential application domains that are more challenging for XCS but depict similar characteristics.

The employed classifier population with a maximum size N of 4,000 seems to be rather large, considering that GAE-LCT [98] used a population with a maximum of 128 classifiers to tackle a similar application scenario, albeit with a different LCS. In our case, smaller maximum population sizes resulted in visible drops in the performance of the frequency control. However, employing large classifier populations can increase the execution time of XCS substantially [13]. This is especially problematic when managing a CPU, as the computational overhead imposed by XCS risks cannibalizing its benefits. Therefore, hardware implementations with low latencies are favorable [116], but larger classifier populations will require more memory resources and bound the latency.

That such a large population is required is striking, as the application scenario can be considered relatively simple. Even a straightforward static strategy achieved good results and sometimes even outperformed XCS. Further, only a tiny fraction of the input state space is actually sampled in the training sets. Using Q-learning with 10 bins, it has been observed that at maximum 0.04 % of the Q-table's entries received an update, i.e., have been activated because a matching input state was observed. Therefore, smaller classifier populations should also be able to cover all occurring input states. However, the unbalanced sampling of input states results in an uneven allocation of classifiers to environmental niches, which prevents the successful use of smaller populations. As seen in the previous subsection, all benchmark applications have some sparsely covered phases with few classifiers in the match set. If the population size

is reduced, these phases might no longer be covered adequately, resulting in random actions. The same effect could be observed if the problem environment becomes more complex and a larger number of different input states is observed, e.g., because user applications are controlled that depict more distinct computational phases than the evaluated benchmark applications. In such cases, the maximum size of the classifier population must be increased, potentially up to the point where the computational overhead of XCS becomes unfeasibly large.

The most severe restriction for employing XCS in autonomous, self-aware systems is likely the non-zero prediction error of XCS. A considerable and oscillating prediction error has been observed for all benchmark applications, both in the overall population and the match set. This prevents the successful use of the error-based E/E strategies investigated in Chapter 4. Due to the creation of specialized classifiers via covering, the prediction error at the beginning of the training process is already close to the minimum achieved by XCS. Hence, the prediction error does not decrease considerably during the learning process, which makes it impossible for error-based E/E strategies to initially apply exploration and then shift toward exploitation. Furthermore, since the error oscillates and can differ considerably between environmental niches, which is best observed for Freqmine in Figure 6.8b, there exists no straight-forward modification for error-based E/E strategies that would still allow their successful deployment. At least the intuitive approach of determining the expected prediction error depicted by a trained population and subtracting it from the observed prediction error is likely infeasible. Hence, autonomous E/E strategies must be employed that do not rely on the prediction error of XCS but instead on performance metrics of the problem environment, e.g., the IPS shortfall in the case of CPU frequency control. Naturally, such strategies must always be tailored to the application domain by the system designers. However, it is noteworthy that even minor differences in the problem environment can impact the behavior of XCS's prediction error, as GAE-LCT [98] successfully employed a local error-based E/E strategy for controlling the CPU frequency, albeit with a different user application.

6.6 CONCLUSION AND FUTURE WORK

In this case study, we have employed XCS for controlling a CPU's frequency to reach a user application's IPS target and minimize power consumption. Our experimental evaluation showed that XCS achieves the best results for two out of three user applications. For one application, it is surpassed by a static heuristic. However, XCS outperforms tabular Q-learning in all experiments. Thereby, we can confirm the observations of related work that LCSs with their dynamic classifier generalization are more suited for performing DVFS than tabular Q-learning with fixed generalizations.

Our investigation of XCS's learning behavior revealed that it behaves differently than in the toy problems commonly used in LCS research. Throughout the experiments, (1) the size of the classifier population does not decrease, indicating the absence of successful classifier generalization, (2) the population's classifiers are not evenly allocated to the environmental niches, and (3) the classifiers' prediction errors do not approach zero. These observed phenomena prevent the successful use of XCS in more complex application environments.

As an investigation of the user applications' execution characteristics showed, the reward is inherently unpredictable to a certain extent, and the input state space is sampled unbalancedly. A temporal pattern has been observed in some cases, where similar states are repeatedly sampled in a sequence. Using theoretical insights on XCS for hyperparameter tailoring, which aims at counteracting some of the adverse effects of the environments' characteristics, has resulted in only minor improvements in learning behavior. In addition, some of the considered approaches, such as a decreased learning rate β or low GA activation frequency, have a detrimental impact on XCS's ability to adapt in the face of environmental dynamics, which have not been considered in this case study.

However, even if a suitable hyperparameter configuration had been found, it would likely apply to only a single user application, as all evaluated applications depict different characteristics. Determining suitable hyperparameters for each individual application scenario by hand is cumbersome and likely infeasible for system designers unfamiliar with XCS – regardless of whether XCS is applied for DVFS or in other application domains. The most promising solution are mechanisms that allow XCS to self-configure its hyperparameters. While some self-configuration approaches already exist, e.g., for the mutation and learning rate [46] or the error threshold [36], they have been mostly designed in isolation and evaluated on toy problem environments only. When developing new hyperparameter self-configuration mechanisms for XCS, future work must (1) investigate characteristics of additional real-world application environments, (2) identify relevant hyperparameters, and (3) design and evaluate the self-configuration mechanisms for multiple hyperparameters in combination to account for possible interactions.

Aside from the hyperparameter configuration, the components of XCS could also be modified. Comparing our case study with GAE-LCT [98] yields indicators that XCS with conditions using discretized intervals can be more robust than XCS with the real-valued intervals that we have employed. With discretized intervals, a classifier is more likely to be subsumed or identified as identical to an existing classifier, resulting in a smaller and more generalized classifier population. Further, in the presence of reward variance, precise condition boundaries likely offer only limited benefits for improving prediction accuracy. Lastly, the use of XCS for environments with a reward function that is largely unpredictable can be questioned in general. Due to the classifier fitness being based on prediction accuracy, XCS will never be able to reliably evolve classifiers

with high fitness, which limits the generalization capability of the GA. Hence, strength-based LCSs could be revisited for use in real-world applications with strong reward variance.

7

CONCLUSION AND FUTURE WORK

Computational self-awareness is a design paradigm for enabling technical systems to autonomously adapt to the specifics of the operational environment and occurring changes, thereby moving design time decisions to the runtime and into the system's responsibility. Learning Classifier Systems (LCSs), especially their most common variant XCS, are frequently proposed for implementing autonomous and adaptive behavior in self-aware systems. However, XCS has been developed and evaluated primarily in artificial toy problem environments, as research on how XCS can be applied in more practical, real-world application environments is lacking. To close the gap in research, this thesis investigated how XCS can be made more amenable for deployment in autonomous and adaptive systems that tackle practical application scenarios. The contributions of this work are threefold:

- Existing autonomous Explore/Exploit (E/E) strategies for XCS from research literature have never been compared to each other, nor has their E/E behavior been extensively investigated. This makes it challenging for XCS practitioners to successfully apply an E/E strategy, which is a necessity for achieving adaptivity. This thesis is the first to experimentally compare four E/E strategies for XCS, revealing that different strategies depict vastly different E/E behaviors. The comparison took place not only in static environments but also in dynamic environments that change during runtime. A local error-based E/E strategy turned out to be most suited for use in self-aware systems, as it depicts a robust E/E behavior, reacts adequately to environmental changes, and is the easiest to parameterize. However, it is unsuited for multi-step environments with sparse rewards. The parameterization has been systematically determined by an automated parameter optimization. Since the optimization also targeted a "one-fits-all" parameterization, XCS practitioners are now equipped with a set of useful strategy parameter configurations.
- *Forbidden classifiers* have been proposed to equip XCS with safety guarantees, which is especially relevant in the context of Cyber-Physical Systems (CPSs). Forbidden classifiers are manually created and injected into XCS's population before deployment. During operation, they prevent the selection of actions that violate safety

requirements. As such, the concept of forbidden classifiers is, to the best of our knowledge, among the first use cases that systematically leverage the interpretability of XCS's rules to inject domain knowledge into the classifier population. Compared to ensuring the safety requirements with an external shield, using forbidden classifiers enables XCS to evolve a solution more quickly, with a smaller classifier population, and reduced computational burden. In addition, adverse effects of overgeneralization are reduced. Thus, forbidden classifiers successfully leverage the classifiers' interpretability to improve the applicability of XCS in application environments requiring safety guarantees.

- In a case study, XCS has been employed to perform Dynamic Voltage and Frequency Scaling (DVFS) and manage the execution of a user application on a CPU. In contrast to the common toy problem environments, this represents a practical, real-world application domain. Even though LCSs have already been applied frequently for DVFS, their learning behavior during operation has, at most, been investigated superficially. This thesis has confirmed the observations of related work that XCS is well-suited for performing DVFS and has also extensively studied the learning behavior of XCS. It was revealed that XCS is adversely affected by an unbalanced sampling of input states and a reward function that is not accurately predictable. Both effects hinder the evolution of adequately generalized classifiers and prevent the successful deployment of XCS in more complex application environments. As a countermeasure, an attempt toward hyperparameter tailoring has been conducted. However, even though the parameterization followed guidelines of existing theoretical works on XCS, it resulted in only mild improvements in XCS's learning behavior. Hence, our case study showed that some of the benefits promised by XCS, such as the evolution of a minimally sized classifier population, might be observable in artificial toy problem environments but not necessarily in real-world application domains.

7.1 FUTURE WORK

Even though this thesis has made significant steps toward making XCS amenable for deployment in autonomous and adaptive systems that tackle practical application scenarios, several open challenges remain. As such, each chapter points to possible future research directions.

- The evaluation of E/E strategies has revealed that the local error-based strategy is most promising but incapable of solving multi-step problems. Thus, it requires modifications to apply to a broader range of problem environments. In addition, our case study has revealed that real-world problem environments can depict unpredictable rewards, which prevents the evolution of accurate classi-

fiers. This highlights the need to develop application-specific E/E strategies that are independent of XCS's prediction error.

- For an effortless use of forbidden classifiers, approaches could be devised that take formalized safety requirements and automatically derive a corresponding, minimal set of forbidden classifiers.
- The use of hand-crafted classifiers to manually inject domain knowledge could be expanded beyond the scope of forbidden classifiers. Instead of preventing the selection of specific actions, the interpretability of classifiers could be leveraged by initializing the population with a set of classifiers representing a known heuristic. However, it is unclear how such classifiers should be treated in the Genetic Algorithm (GA) and during action selection. Initially, they should dominate the behavior of XCS to enforce the heuristic, but at later stages of learning, they have to allow the evolution of a better solution.
- Future work must investigate mechanisms that automatically detect an unbalanced sampling of the input state space or other adverse phenomena and take appropriate countermeasures. This includes hyperparameter self-configuration approaches, which would also facilitate the use of XCS by system designers that are unfamiliar with XCS and cannot optimize hyperparameters by hand.
- XCS must be made more robust against reward variance for deployment in real-world environments, as existing approaches are often applicable to artificial toy problem environments only (cf. Section 3.3). That the degree of reward variance differs between states could pose a significant challenge to the development of such approaches. An outcome could be the use of hybrid fitness functions that consider both the accuracy and magnitude of the payoff prediction. An early attempt at using such a hybrid fitness function has been made in [14]. However, in the presence of strong reward variance, it can be argued that accuracy-based fitness, as used in XCS, is inherently unsuited. Hence, the use of strength-based LCSs could be revisited, potentially with extensions that alleviate their disadvantages, which have been discussed in Section 3.2.

BIBLIOGRAPHY

[1] Andreas Agne, Markus Happe, Achim Lösch, Christian Plessl, and Marco Platzner. "Self-Awareness as a Model for Designing and Operating Heterogeneous Multicores." In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.2 (2014), pp. 1–18. DOI: 10.1145/2617596.

[2] Adam A. Alli and Muhammad Mahbub Alam. "The fog cloud of things: A survey on concepts, architecture, standards, tools, and applications." In: *Internet of Things* 9 (2020). DOI: 10.1016/J.IOT.2020.100177.

[3] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. "Safe Reinforcement Learning via Shielding." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2018, pp. 2669–2678.

[4] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. "Concrete Problems in AI Safety." In: *CoRR* abs/1606.06565 (2016). arXiv: 1606.06565.

[5] *An architectural blueprint for autonomic computing*. Tech. rep. IBM, 2005. URL: https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.

[6] Jaume Bacardit, Edmund K Burke, and Natalio Krasnogor. "Improving the scalability of rule-based evolutionary learning." In: *Memetic Computing* 1.1 (2009), pp. 55–67. DOI: 10.1007/s12293-008-0005-4.

[7] Thomas Back and Hans-Paul Schwefel. "Evolutionary computation: an overview." In: *Proceedings of IEEE International Conference on Evolutionary Computation*. 1996, pp. 20–29. DOI: 10.1109/ICEC.1996.542329.

[8] Anthony J. Bagnall and George D. Smith. "A multiagent model of the UK market in electricity generation." In: *IEEE Transactions on Evolutionary Computation* 9.5 (2005), pp. 522–536. DOI: 10.1109/TEVC.2005.850264.

[9] K. Bellman et al. "Self-aware Cyber-Physical Systems." In: *ACM Transactions on Cyber-Physical Systems* 4.4 (2020), pp. 1–26. DOI: 10.1145/3375716.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC benchmark suite: Characterization and architectural implications." In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* (2008), pp. 72–81. DOI: 10.1145/1454115.1454128.

[11] Nathan Binkert et al. "The gem5 simulator." In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[12] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Springer, 1984. ISBN: 978-1-4612-9784-0. DOI: 10.1007/978-1-4613-2821-6.

[13] Mathis Brede, Tim Hansmeier, and Marco Platzner. "XCS on embedded systems: an analysis of execution profiles and accelerated classifier deletion." In: *Proceedings of the 2022 Genetic and Evolutionary Computation Conference Companion (GECCO '22)*. ACM, 2022, pp. 2071–2079. DOI: 10.1145/3520304.3533977.

[14] William Browne. "The Development of an Industrial Learning Classifier System for Data-Mining in a Steel Hop Strip Mill." In: *Applications of Learning Classifier Systems*. Springer Berlin Heidelberg, 2004, pp. 223–259. ISBN: 978-3-540-39925-4. DOI: 10.1007/978-3-540-39925-4_10.

[15] Larry Bull and Toby O'Hara. "Accuracy-Based Neuro and Neuro-Fuzzy Classifier Systems." In: *Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02)*. 2002, pp. 905–911.

[16] Alwyn Burger, David W. King, and Gregor Schiele. "Reconfigurable embedded devices using reinforcement learning to develop action-policies." In: *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2020)*. 2020, pp. 232–241. DOI: 10.1109/ACSOS49614.2020.00046.

[17] M. V. Butz and S. W. Wilson. "An algorithmic description of XCS." In: *Soft Computing* 6.3-4 (2002), pp. 144–153. DOI: 10.1007/s005000100111.

[18] Martin V Butz. *Anticipatory learning classifier systems*. Springer New York, 2002. ISBN: 978-0-7923-7630-9. DOI: 10.1007/978-1-4615-0891-5.

[19] Martin V. Butz. "Ellipsoidal conditions in the real-valued XCS classifier system." In: *Proceedings of the 2005 Genetic and Evolutionary Computation Conference (GECCO '05)*. 2005, pp. 1835–1842. DOI: 10.1145/1068009.1068320.

[20] Martin V. Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W. Wilson. "Toward a Theory of Generalization and Learning in XCS." In: *IEEE Transactions on Evolutionary Computation* 8.1 (2004), pp. 28–46. DOI: 10.1109/TEVC.2003.818194.

[21] Martin V. Butz, Pier Luca Lanzi, and Stewart W. Wilson. "Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction." In: *IEEE Transactions on Evolutionary Computation* 12.3 (2008), pp. 355–376. DOI: 10.1109/TEVC.2007.903551.

[22] Arjun Chandra, Peter R. Lewis, Kyrre Glette, and Stephan C. Stilkerich. "Reference Architecture for Self-aware and Self-expressive Computing Systems." In: *Self-aware Computing Systems: An Engineering Approach*. Springer International Publishing, 2016, pp. 37–49. ISBN: 978-3-319-39675-0. DOI: 10.1007/978-3-319-39675-0_4.

[23] Tao Chen, Funmilade Faniyi, and Rami Bahsoon. "Design patterns and primitives: Introduction of components and patterns for SACS." In: *Self-aware Computing Systems: An Engineering Approach*. Springer International Publishing, 2016, pp. 53–78. DOI: 10.1007/978-3-319-39675-0_5.

[24] Janez Demšar. "Statistical Comparisons of Classifiers over Multiple Data Sets." In: *Journal of Machine Learning Research* 7 (2006), pp. 1–30.

[25] Simon Dobson, Roy Sterritt, Paddy Nixon, and Mike Hinchey. "Fulfilling the vision of autonomic computing." In: *Computer* 43.1 (2010), pp. 35–41. DOI: 10.1109/MC.2010.14.

[26] Bryan Donyanavard, Tiago Muck, Amir M. Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. "SOSA: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management." In: *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '52)*. 2019, pp. 685–698. DOI: 10.1145/3352460.3358312.

[27] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. "SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores." In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES '16)*. 2016. DOI: 10.1145/2968456.2968459.

[28] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: http://archive.ics.uci.edu/ml.

[29] Nikil Dutt, Fadi J. Kurdahi, Rolf Ernst, and Andreas Herkersdorf. "Conquering MPSoC complexity with principles of a self-aware information processing factory." In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'16)*. 2016, pp. 1–4.

[30] Melanie Feist, Martin Breitbach, Heiko Trotsch, Christian Becker, and Christian Krupitzer. "Rango: An Intuitive Rule Language for Learning Classifier Systems in Cyber-Physical Systems." In: *Proceedings of the 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2022)*. 2022, pp. 31–40. ISBN: 9781665471374. DOI: 10.1109/ACSOS55765.2022.00021.

[31] Milton Friedman. "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance." In: *Journal of the American Statistical Association* 32.200 (1937), pp. 675–701. DOI: 10.1080/01621459.1937.10503522.

[32] Javier García and Fernando Fernández. "A Comprehensive Survey on Safe Reinforcement Learning." In: *Journal of Machine Learning Research* 16.42 (2015), pp. 1437–1480.

[33] Ujjwal Gupta, Sumit K. Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y. Ogras. "A Deep Q-Learning Approach for Dynamic Management of Heterogeneous Processors." In: *IEEE Computer Architecture Letters* 18.1 (2019), pp. 14–17. DOI: 10.1109/LCA.2019.2892151.

[34] Ali Hamzeh and Adel Rahmani. "A Fuzzy System to Control Exploration Rate in XCS." In: *International Workshop on Learning Classifier Systems (IWLCS 2003-2005)*. 2005, pp. 115–127. DOI: 10.1007/978-3-540-71231-2_9.

[35] Tim Hansmeier, Mathis Brede, and Marco Platzner. "Safe Learning with XCS via the Injection of Forbidden Classifiers." In: *SN Computer Science* (tbd). Currently under review.

[36] Tim Hansmeier, Paul Kaufmann, and Marco Platzner. "An Adaption Mechanism for the Error Threshold of XCSF." In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*. 2020, pp. 1756–1764. DOI: 10.1145/3377929.3398106.

[37] Tim Hansmeier and Marco Platzner. "An experimental comparison of explore/exploit strategies for the learning classifier system XCS." In: *Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21)*. 2021, pp. 1639–1647. ISBN: 9781450383516. DOI: 10.1145/3449726.3463159.

[38] Tim Hansmeier and Marco Platzner. "Integrating Safety Guarantees into the Learning Classifier System XCS." In: *Applications of Evolutionary Computation (EvoApplications 2022)*. Springer, 2022, pp. 386–401. DOI: 10.1007/978-3-031-02462-7_25.

[39] Tim Hansmeier and Marco Platzner. "Autonomous Explore/Exploit Strategies for XCS: An Experimental Comparison." In: *Soft Computing* (tbd). Currently under review.

[40] Adrian R. Hartley. "Accuracy-based fitness allows similar performance to humans in static and dynamic classification environments." In: *Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO '99)*. 1999, pp. 266–273.

[41] Michael Heider, David Pätzel, and Alexander R.M. Wagner. "An overview of LCS research from 2021 to 2022." In: *Proceedings of the 2022 Genetic and Evolutionary Computation Conference Companion (GECCO '22)*. 2022, pp. 2086–2094. DOI: 10.1145/3520304.3533985.

[42] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments." In: *Proceedings of the 7th International Conference on Autonomic Computing (ICAC '10) and Co-located Workshops*. 2010, pp. 79–88. DOI: 10.1145/1809049.1809065.

[43] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. *SEEC: A General and Extensible Framework for Self-Aware Computing*. Tech. rep. MIT-CSAIL-TR-2011-046. 2011. URL: http://hdl.handle.net/1721.1/67020.

[44] Henry Hoffmann et al. "Self-aware computing in the Angstrom processor." In: *Proceedings of the Design Automation Conference (DAC '12)* (2012), pp. 259–264. DOI: 10.1145/2228360.2228409.

[45] Sture Holm. "A Simple Sequentially Rejective Multiple Test Procedure." In: *Scandinavian Journal of Statistics* 6.2 (1979), pp. 65–70.

[46] Jacob Hurst and Larry Bull. "A self-adaptive XCS." In: *International Workshop on Learning Classifier Systems (IWLCS 2001)*. 2002, pp. 57–73. DOI: 10.1007/3-540-48104-4_5.

[47] Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. "Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems." In: *IEEE Transactions on Evolutionary Computation* 18.4 (2014), pp. 465–480. DOI: 10.1109/TEVC.2013.2281537.

[48] Arman Iranfar, Soheil Nazar Shahsavani, Mehdi Kamal, and Ali Afzali-Kusha. "A heuristic machine learning-based algorithm for power and thermal management of heterogeneous MPSoCs." In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2015, pp. 291–296. DOI: 10.1109/ISLPED.2015.7273529.

[49] Jeffrey O. Kephart and David M. Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[50] Tim Kovacs. "Deletion Schemes for Classifier Systems." In: *Proceedings of the 1999 Conference on Genetic and Evolutionary Computation (GECCO '99)*. 1999, pp. 329–336.

[51] Tim Kovacs. "Strength or Accuracy? Fitness Calculation in Learning Classifier Systems." In: *International Workshop on Learning Classifier Systems (IWLCS 1999)*. Springer Berlin Heidelberg, 1999, pp. 143–160. DOI: 10.1007/3-540-45027-0_7.

[52] Tim Kovacs. "Performance and population state metrics for rule-based learning systems." In: *Proceedings of the 2002 Congress on Evolutionary Computation (CEC '02)*. 2002, pp. 1781–1786. DOI: 10.1109/CEC.2002.1004512.

[53] Tim Kovacs. "Strength or accuracy: credit assignment in learning classifier systems." PhD thesis. Bristol, Univ., 2004. ISBN: 1-85233-770-2.

[54] Philippe Lalanda, Julie A McCann, and Ada Diaconescu. *Autonomic Computing - Principles, Design and Implementation*. Springer, 2013. ISBN: 978-1-4471-5006-0. DOI: 10.1007/978-1-4471-5007-7.

[55] P. L. Lanzi and S. W. Wilson. "Toward Optimal Classifier System Performance in Non-Markov Environments." In: *Evolutionary Computation* 8.4 (2000), pp. 393–418. DOI: 10.1162/106365600568239.

[56] Pier Luca Lanzi. "A Study of the Generalization Capabilities of XCS." In: *Proceedings of the 7th International Conference on Genetic Algorithms*. 1997, pp. 418–425.

[57] Pier Luca Lanzi. "Adding memory to XCS." In: *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC)* (1998), pp. 609–614. DOI: 10.1109/ICEC.1998.700098.

[58] Pier Luca Lanzi. "An Analysis of Generalization in the XCS Classifier System." In: *Evolutionary Computation* 7.2 (1999), pp. 125–149. DOI: 10.1162/evco.1999.7.2.125.

[59] Pier Luca Lanzi. "An Analysis of the Memory Mechanism of XCSM." In: *Genetic Programming 98*. 1999, pp. 643–651.

[60] Pier Luca Lanzi and Marco Colombetti. "An Extension to the XCS Classifier System for Stochastic Environments." In: *Proceedings of the 1999 Conference on Genetic and Evolutionary Computation (GECCO '99)*. 1999, pp. 353–360.

[61] Pier Luca Lanzi and Daniele Loiacono. "XCSF with neural prediction." In: *Proceedings of the 2006 Congress on Evolutionary Computation (CEC '06)*. 2006, pp. 2270–2276. DOI: 10.1109/CEC.2006.1688588.

[62] Pier Luca Lanzi, Daniele Loiacono, Siewart W. Wilson, and David E. Goldberg. "Extending XCSF Beyond Linear Approximation." In: *Proceedings of the 2005 Genetic and Evolutionary Computation Conference (GECCO '05)*. 2005, pp. 1827–1834. DOI: 10.1145/1068009.1068319.

[63] Veronika Lesch, Tanja Noack, Johannes Hefter, Samuel Kounev, and Christian Krupitzer. "Towards Situation-Aware Meta-Optimization of Adaptation Planning Strategies." In: *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2021)*. 2021, pp. 177–187. DOI: 10.1109/ACSOS52086.2021.00042.

[64] Peter R. Lewis, Arjun Chandra, Funmilade Faniyi, Kyrre Glette, Tao Chen, Rami Bahsoon, Jim Torresen, and Xin Yao. "Architectural aspects of self-Aware and self-expressive computing systems: From psychology to engineering." In: *Computer* 48.8 (2015), pp. 62–70. DOI: 10.1109/MC.2015.235.

[65] Peter R. Lewis, Arjun Chandra, and Kyrre Glette. "Self-awareness and Self-expression: Inspiration from Psychology." In: *Self-aware Computing Systems: An Engineering Approach*. Springer International Publishing, 2016, pp. 9–21. ISBN: 978-3-319-39675-0. DOI: 10.1007/978-3-319-39675-0_2.

[66] Peter R. Lewis, Marco Platzner, Bernhard Rinner, Jim Tørresen, and Xin Yao, eds. *Self-aware Computing Systems: An Engineering Approach*. Springer International Publishing, 2016. ISBN: 978-2-319-39675-0. DOI: 10.1007/978-3-319-39675-0.

[67] Lei Liu, Stefan Thanheiser, and Hartmut Schmeck. "Assessing the impact of inherent SOA system properties on complexity." In: *Proceedings of the 4th International Conference on Internet and Web Applications and Services (ICIW 2009)*. 2009, pp. 429–434. DOI: 10.1109/ICIW.2009.70.

[68] Shiting Lu, Russell Tessier, and Wayne Burleson. "Reinforcement learning for thermal-aware many-core task allocation." In: *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI 2015)*. 2015, pp. 379–384. DOI: 10.1145/2742060.2742078.

[69] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. "The irace package: Iterated Racing for Automatic Algorithm Configuration." In: *Operations Research Perspectives* 3 (2016), pp. 43–58. DOI: 10.1016/j.orp.2016.09.002.

[70] Florian Maurer, Bryan Donyanavard, Amir M. Rahmani, Nikil Dutt, and Andreas Herkersdorf. "Emergent Control of MPSoC Operation by a Hierarchical Supervisor / Reinforcement Learning Approach." In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 2020)*. 2020, pp. 1562–1567. DOI: 10.23919/DATE48585.2020.9116574.

[71] Alex McMahon, Dan Scott, and Will Browne. "An Autonomous Explore/Exploit Strategy." In: *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation (@GECCO '05)*. 2005, pp. 103–108. DOI: 10.1145/1102256.1102280.

[72] Christian Müller-Schloer and Sven Tomforde. "Building Organic Computing Systems." In: *Organic Computing – Technical Systems for Survival in the Real World*. Birkhäuser, 2017, pp. 171–258. DOI: 10.1007/978-3-319-68477-2_5.

[73] Christian Müller-Schloer and Sven Tomforde. *Organic Computing – Technical Systems for Survival in the Real World*. Birkhäuser, 2017. ISBN: 978-3-319-68476-5. DOI: 10.1007/978-3-319-68477-2.

[74] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, eds. *Organic Computing — A Paradigm Shift for Complex Systems*. Birkhäuser, 2011. ISBN: 978-3-0348-0129-4. DOI: 10.1007/978-3-0348-0130-0.

[75] Masaya Nakata and Will N. Browne. "Learning optimality theory for accuracy-based learning classifier systems." In: *IEEE Transactions on Evolutionary Computation* 25.1 (2021), pp. 61–74. DOI: 10.1109/TEVC.2020.2994314.

[76] Ulric Neisser. "The Roots of Self-Knowledge: Perceiving Self, It, and Thou." In: *Annals of the New York Academy of Sciences* 818.1 (1997), pp. 19–33. DOI: 10.1111/J.1749-6632.1997.TB48243.X.

[77] Albert Orriols-Puig, Ester Bernadó-Mansilla, David E. Goldberg, Kumara Sastry, and Pier Luca Lanzi. "Facetwise analysis of XCS for problems with class imbalances." In: *IEEE Transactions on Evolutionary Computation* 13.5 (2009), pp. 1093–1119. DOI: 10.1109/TEVC.2009.2019829.

[78] Santiago Pagani, P. D.Sai Manoj, Axel Jantsch, and Jörg Henkel. "Machine Learning for Power, Energy, and Thermal Management on Multicore Processors: A Survey." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.1 (2020), pp. 101–116. DOI: 10.1109/TCAD.2018.2878168.

[79] Manish Parashar and Salim Hariri. "Autonomic computing: An overview." In: *International Workshop on Unconventional Programming Paradigms (UPP 2004)*. Springer Verlag, 2005, pp. 257–269. DOI: 10.1007/11527800_20.

[80] David Pätzel, Michael Heider, and Alexander R. M. Wagner. "An Overview of LCS Research from 2020 to 2021." In: *Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21)*. 2021, pp. 1648–1656. DOI: 10.1145/3449726.3463173.

[81] Tom Pickering and Tim Kovacs. "TP-XCS: An XCS classifier system with fixed-length memory for reinforcement learning." In: *Proceedings of the 2015 Congress on Evolutionary Computation (CEC '15)*. 2015, pp. 3020–3025. DOI: 10.1109/CEC.2015.7257265.

[82] Holger Prothmann, Jürgen Branke, Hartmut Schmeck, Sven Tomforde, Fabian Rochner, Jörg Hähner, and Christian Müller-Schloer. "Organic traffic light control for urban road networks." In: *International Journal of Autonomous and Adaptive Communications Systems* 2.3 (2009), pp. 203–225. DOI: 10.1504/IJAACS.2009.026783.

[83] Holger Prothmann, Sven Tomforde, Jürgen Branke, Jörg Hähner, Christian Müller-Schloer, and Hartmut Schmeck. "Organic Traffic Control." In: *Organic Computing — A Paradigm Shift for Complex Systems*. Birkhäuser, 2011, pp. 431–446. ISBN: 978-3-0348-0129-4. DOI: 10.1007/978-3-0348-0130-0_28.

[84] Eberle A. Rambo et al. "The Self-Aware Information Processing Factory Paradigm for Mixed-Critical Multiprocessing." In: *IEEE Transactions on Emerging Topics in Computing* 10.1 (2022), pp. 250–266. DOI: 10.1109/TETC.2020.3011663.

[85] Lilia Rejeb, Zahia Guessoum, and Rym M'Hallah. "An Adaptive Approach for the Exploration-Exploitation Dilemma and Its Application to Economic Systems." In: *First International Workshop on Learning and Adaption in Multi-Agent Systems (LAMAS 2005)*. 2005, pp. 165–176. DOI: 10.1007/11691839_10.

[86] Bernhard Rinner, Lukas Esterle, Jennifer Simonjan, Georg Nebehay, Roman Pflugfelder, Gustavo Fernandez Dominguez, and Peter R. Lewis. "Self-Aware and Self-Expressive camera networks." In: *Computer* 48.7 (2015), pp. 21–28. DOI: 10.1109/MC.2015.209.

[87] Lukas Rosenbauer, David Pätzel, Anthony Stein, and Jörg Hähner. "A Learning Classifier System for Automated Test Case Prioritization and Selection." In: *SN Computer Science* 3.5 (2022), pp. 1–24. DOI: 10.1007/S42979-022-01255-1.

[88] William Saunders, Andreas Stuhlmüller, Girish Sastry, and Owain Evans. "Trial without error: Towards safe reinforcement learning via human intervention." In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2018)*. 2018, pp. 2067–2069. arXiv: 1707.05173.

[89] Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. "Adaptivity and Self-organisation in Organic Computing Systems." In: *Organic Computing — A Paradigm Shift for Complex Systems*. Birkhäuser, 2011, pp. 5–37. ISBN: 978-3-0348-0129-4. DOI: 10.1007/978-3-0348-0130-0_1.

[90] Hao Shen, Jun Lu, and Qinru Qiu. "Learning based DVFS for simultaneous temperature, performance and energy management." In: *Proceedings of the 2012 International Symposium on Quality Electronic Design (ISQED 2012)*. 2012, pp. 747–754. DOI: 10.1109/ISQED.2012.6187575.

[91] Nora Sperling et al. “Information Processing Factory 2.0 - Self-awareness for Autonomous Collaborative Systems.” In: *Design, Automation & Test in Europe Conference & Exhibition (DATE 2023)*. 2023. DOI: 10.23919/DATE56975.2023.10137006.

[92] Anthony Stein, Roland Maier, and Jörg Hähner. “Toward curious learning classifier systems: Combining XCS with active learning concepts.” In: *Proceedings of the 2017 Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. 2017, pp. 1349–1356. DOI: 10.1145/3067695.3082488.

[93] Anthony Stein, Roland Maier, Lukas Rosenbauer, and Jorg Hahner. “XCS classifier system with experience replay.” In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. 2020, pp. 404–413. DOI: 10.1145/3377930.3390249.

[94] Anthony Stein and Masaya Nakata. “Learning classifier systems: From principles to modern systems.” In: *Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21)*. 2021, pp. 498–527. DOI: 10.1145/3449726.3461414.

[95] Anthony Stein, Dominik Rauh, Sven Tomforde, and Jörg Hähner. “Interpolation in the eXtended Classifier System: An architectural perspective.” In: *Journal of Systems Architecture* 75 (2017), pp. 79–94. DOI: 10.1016/J.SYSARC.2017.01.010.

[96] Anthony Stein, Stefan Rudolph, Sven Tomforde, and Jörg Hähner. “Self-learning smart cameras harnessing the generalization capability of XCS.” In: *Proceedings of the 9th International Joint Conference on Computational Intelligence (IJCCI 2017)*. 2017, pp. 129–140. DOI: 10.5220/0006512101290140.

[97] Christopher Stone and Larry Bull. “For Real! XCS with Continuous-Valued Inputs.” In: *Evolutionary Computation* 11.3 (2003), pp. 299–336. DOI: 10.1162/106365603322365315.

[98] Anmol Surhonne, Nguyen Anh Vu Doan, Florian Maurer, Thomas Wild, and Andreas Herkersdorf. “GAE-LCT: A Run-Time GA-Based Classifier Evolution Method for Hardware LCT Controlled SoC Performance-Power Optimization.” In: *Proceedings of the 2022 International Conference on Architecture of Computing Systems (ARCS 2022)*. 2022, pp. 271–285. DOI: 10.1007/978-3-031-21867-5_18.

[99] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introdcution (2nd ed.)* MIT Press, 2018. ISBN: 978-0262039246.

[100] Takato Tatsumi, Takahiro Komine, Hiroyuki Sato, and Keiki Takadama. “Handling different level of unstable reward environment through an estimation of reward distribution in XCS.” In: *Proceedings of the 2015 Congress on Evolutionary Computation (CEC '15)*. 2015, pp. 2973–2980. DOI: 10.1109/CEC.2015.7257259.

[101] Takato Tatsumi, Tim Kovacs, and Keiki Takadama. "XCS-CR: Determining accuracy of classifier by its collective reward in action set toward environment with action noise." In: *Proceedings of the 2018 Genetic and Evolutionary Computation Conference Companion (GECCO '18)*. 2018, pp. 1457–1464. DOI: 10.1145/3205651.3208271.

[102] Takato Tatsumi, Hiroyuki Sato, and Keiki Takadama. "Automatic adjustment of selection pressure based on range of reward in learning classifier system." In: *Proceedings of the 2017 Genetic and Evolutionary Computation Conference (GECCO '17)*. 2017, pp. 505–512. DOI: 10.1145/3071178.3080531.

[103] Takato Tatsumi and Keiki Takadama. "XCS-CR for handling input, output, and reward noise." In: *Proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. 2019, pp. 1303–1311. DOI: 10.1145/3319619.3326863.

[104] Sven Tomforde, Andreas Brameshuber, Jörg Hahner, and Christian Müller-Schloer. "Restricted on-line learning in real-world systems." In: *Proceedings of the 2011 Congress on Evolutionary Computation (CEC '11)*. 2011, pp. 1628–1635. DOI: 10.1109/CEC.2011.5949810.

[105] Christopher Umans, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. "Complexity of two-level logic minimization." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.7 (2006), pp. 1230–1246. DOI: 10.1109/TCAD.2005.855944.

[106] Ryan J. Urbanowicz and Will N. Browne. *Introduction to Learning Classifier Systems*. Springer Berlin Heidelberg, 2017. ISBN: 978-3-662-55006-9. DOI: 10.1007/978-3-662-55007-6.

[107] Ryan J. Urbanowicz and Jason H. Moore. "ExSTraCS 2.0: Description and Evaluation of a Scalable Learning Classifier System." In: *Evolutionary intelligence* 8.2 (2015), pp. 89–116. DOI: 10.1007/s12065-015-0128-8.

[108] Matthew Walker, Sascha Bischoff, S. Diestelhorst, Geoff Merrett, and Bashir Al-Hashimi. "Hardware-Validated CPU Performance and Energy Modelling." In: *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2018)*. 2018, pp. 44–53. DOI: 10.1109/ISPASS.2018.00013.

[109] Shuo Wang, Georg Nebehay, Lukas Esterle, Kristian Nymoen, and Leandro L. Minku. "Common techniques for self-awareness and self-expression." In: *Self-aware Computing Systems: An Engineering Approach*. Springer International Publishing, 2016, pp. 113–142. ISBN: 978-3-319-39675-0. DOI: 10.1007/978-3-319-39675-0_7.

[110] Stewart W. Wilson. "Classifier Fitness Based on Accuracy." In: *Evolutionary Computation* 3.2 (1995), pp. 149–175. DOI: 10.1162/evco.1995.3.2.149.

[111] Stewart W. Wilson. "Explore/Exploit Strategies in Autonomy." In: *Proceedings of the 4th International Conference on Simulation of Adaptive Behavior*. 1996, pp. 325–332. DOI: 10.7551/mitpress/3118.003.0040.

[112] Stewart W. Wilson. "Generalization in the XCS Classifier System." In: *Proceedings of the Third Annual Genetic Programming Conference*. 1998, pp. 665–674.

[113] Stewart W Wilson. "Get Real! XCS with Continuous-Valued Inputs." In: *International Workshop on Learning Classifier Systems (IWLCS 2000)*. 2000, pp. 209–219.

[114] Stewart W. Wilson. "Classifiers that approximate functions." In: *Natural Computing* 1.2 (2002), pp. 211–234. DOI: 10.1023/A:1016535925043.

[115] Zhaoxiang Zang, Dehua Li, and Junying Wang. "Learning classifier systems with memory condition to solve non-Markov problems." In: *Soft Computing* 19.6 (2015), pp. 1679–1699. DOI: 10.1007/S00500-014-1357-Y.

[116] J Zeppenfeld, A Bouajila, W Stechele, and A Herkersdorf. "Learning Classifier Tables for Autonomic Systems on Chip." In: *INFORMATIK 2008. Beherrschbare Systeme - dank Informatik. Band 2*. Gesellschaft für Informatik e. V., 2008, pp. 771–778.

[117] Johannes Zeppenfeld and Andreas Herkersdorf. "Applying Autonomic Principles for Workload Management in Multi-Core Systems on Chip." In: *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011)*. 2011, pp. 3–10. DOI: 10.1145/1998582.1998586.

[118] Robert F Zhang and Ryan J Urbanowicz. "A Scikit-learn Compatible Learning Classifier System." In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*. 2020. DOI: 10.1145/3377929.3398097.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/