

ConCS: A Continual Classifier System for Continual Learning of Multiple Boolean Problems

Trung B. Nguyen¹, *Member, IEEE*, Will N. Browne², *Member, IEEE*, and Mengjie Zhang³, *Fellow, IEEE*

Abstract—Human intelligence can simultaneously process many tasks with the ability to accumulate and reuse knowledge. Recent advances in artificial intelligence, such as transfer, multitask, and layered learning, seek to replicate these abilities. However, humans must specify the task order, which is often difficult particularly with uncertain domain knowledge. This work introduces a continual-learning system (ConCS), such that given an open-ended set of problems once each is solved its solution can contribute to solving further problems. The hypothesis is that the evolutionary computation approach of learning classifier systems (LCSs) can form this system due to its niched, cooperative rules. A collaboration of parallel LCSs identifies sets of patterns linking features to classes that can be reused in related problems automatically. Results from distinct Boolean and integer classification problems, with varying interrelations, show that by combining knowledge from simple problems, complex problems can be solved at increasing scales. 100% accuracy is achieved for the problems tested regardless of the order of task presentation. This includes intractable problems for previous approaches, e.g., n -bit Majority-on. A major contribution is that human guidance is now unnecessary to determine the task learning order. Furthermore, the system automatically generates the curricula for learning the most difficult tasks.

Index Terms—Building blocks, code fragment (CF), continual learning, learning classifier systems (LCS), multitask learning (MTL).

I. INTRODUCTION

ACCUMULATING and transferring/reusing knowledge are inherent abilities of humans. A human accumulates knowledge through their lifetime from the most intuitive concepts and simple skills to increasingly abstract and complex knowledge [1]. This increasing order of knowledge difficulty is important for fast learning progress, but humans do not need

strict orders of problems, skills, and lessons to progressively acquire knowledge.

The abilities of knowledge accumulation and reusability are also desirable features for artificial intelligence (AI) systems [2], [3] as learning complex problems from scratch faces the challenge of intractable search spaces. Layered learning (LL) is a sequential learning paradigm that can achieve such abilities [4]. LL enables learning complex knowledge, plus functions to manipulate this knowledge, by incrementally learning a series of subtasks and associated component knowledge, where previous knowledge, i.e., functions and skills,¹ can bootstrap later tasks. Similarly, continual learning is an AI concept that encapsulates the continuity of the sequential learning process with knowledge reusability [5]. These mechanisms are analogous to how humans accumulate knowledge, as mentioned above. LL normally refers to sequential LL by default, which assimilates knowledge components sequentially. Sequential learning requires human guidance to specify an order of learning tasks (learning order) that allows each learning stage to obtain its target knowledge. Also, it limits the autonomy of the AI system. Automatic discovery of the learning order is also a desirable ability for an AI system because it provides understandings on the dependencies among knowledge components. This feature leads to concurrent LL, which is learning all stages of sequential LL in parallel without the requirement of the learning order [6], [7]. However, a concurrent LL system targets only a specific problem. This limits the potential of such systems to be within specific domains.

In this work, we seek to utilize knowledge reusability in a system of multiple distributed learning agents² to accumulate knowledge and solve multiple problems for the following benefits. First, the intractable search space of a complex problem can be divided amongst agents to ease the task. By solving more problems, the system can accumulate more knowledge and maximize its problem-solving capability. Second, having multiple agents can minimize the complexity of each agent, which arguably encourages the generality of each agent. Accordingly, the system can flexibly adapt its complexity during its operation by adding or removing agents. Finally, this system can work as a continuous learning system [8], i.e., designed for problem solving—especially pattern discovery,

Manuscript received 15 November 2021; revised 16 May 2022 and 12 August 2022; accepted 5 September 2022. Date of publication 4 October 2022; date of current version 1 August 2023. (Corresponding author: Trung B. Nguyen.)

Trung B. Nguyen and Mengjie Zhang are with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington 6140, New Zealand (e-mail: trungnbao@gmail.com; mengjie.zhang@ecs.vuw.ac.nz).

Will N. Browne is with the Faculty of Engineering, Queensland University of Technology, Brisbane, QLD 4000, Australia (e-mail: will.browne@qut.edu.au).

This article has supplementary material provided by the authors and color versions of one or more figures available at <https://doi.org/10.1109/TEVC.2022.3210872>.

Digital Object Identifier 10.1109/TEVC.2022.3210872

¹A skill is a function without a returned value, e.g., the loop skill.

²In the scope of this article, an agent is a complete evolutionary computation (EC) system.

and state-action-reward/state-class predictions—from a stream of data from various problems.

EC algorithms can learn optimization problems through building blocks that can facilitate sharing knowledge [9]. There are many attempts to exploit this feature of EC to develop EC algorithms with the ability of reusing knowledge across multiple tasks. One of the research directions in EC focused on this ability is evolutionary multitasking [10]. This research direction is related to our work in terms of the distribution property, but such algorithms are mainly designed for optimization, rather than the classification tasks considered here [11].

Among EC algorithms, learning classifier systems (LCSs) are a problem-solving approach that is decentralized by nature by dividing a problem into subproblems, which can be solved more easily by subsets of its solution [12]. XCS is the most common LCS, with its accuracy-based Michigan approach [13] having been developed and applied to a wide range of tasks since 1994 [14], [15]. The use of code fragments (CFs), directed graphs that have input data features or other CFs in their leaf nodes and functions in internal nodes to form tree-based programs, in LCSs has extended their scalability, particularly by enabling knowledge reusability. This ability makes CF-based LCSs promising algorithms for imitating human-learning abilities. XCSCFC enabled feature reusability where relevant building blocks of knowledge was transferred in the form of CFs [9]. Alvarez et al. showed that an LCS-produced ruleset could be treated as a function,³ which is reusable in future tasks [16]. He then extended the reusability of ruleset functions to produce XCSCF* [17] with LL. This system was able to discover the complicated logic of the multiplexer (Mux) problem domain, so can solve the problem at any given scale. However, these attempts at implementing the ability to reuse knowledge were limited to sequential learning. Also, they required human intervention, such as specifying the numbers of transferred CFs and designing a curriculum.⁴ Designing a curriculum is a nontrivial task because it requires a deep understanding of target problems in advance [18]. Therefore, these systems were strictly limited in flexibility and not appropriate for a continual learning system.

A justification for the use of LCSs is the problem of “catastrophic forgetting,” which is common in machine learning, where once trained on one task and then trained on a second task, many machine learning models forget how to perform the first task [19]. In connectionist approaches to machine learning, this can be attributed to overfitting (hence using dropout and other regularization techniques to mitigate the problem) [20] and storing knowledge in a network that is likened to a locality-sensitive hashtable [21] (hence considering capsule networks to increase spatial information [22]). LCSs are inherently symbolic (so can reason about variables directly), built on the concept of generalization (e.g., schema theory and the “don’t care” representation concept to avoid overfitting) [23] and are niche-based (subsets of rules only

address specific regions of the input space, so can be kept until reencountered) [24].

In this study, we propose a novel continuous AI system of multiple XCS-based agents, termed continual classifier system (ConCS). ConCS is targeted to learn multiple tasks in parallel and continuously. ConCS solves problems with the ability to accumulate knowledge in a knowledge pool and reuse it in novel tasks. This results in its capability of tackling complex problems at the scale that no other AI systems can without *a priori* knowledge.

The contribution is a novel system that can be presented with any Boolean string-encoded classification/regression problem(s) at any time. ConCS will discover any relationship to previous problems to bootstrap learning, solve the problem in a continuous learning manner, and fit the problem into its developing curricula. The novel methods in ConCS will enable it to encode learned knowledge k_i using CF-based rule-set functions ($f(s)$) in a knowledge pool $[KP] = \{k_i\} + \{f_i\}$. These methods address different tasks using the introduced connected and distributed representation as $[KP]$. This avoids the requirement to sum all knowledge learned from past tasks $[KP] \neq \sum_{i \in \text{tasks}} (k_i)$, which may result in a forgetting effect like the catastrophic forgetting in connectionist approaches [25]. Type-fitting based on the representation is introduced to further reduce the size of $[KP]$, which is essential for computation efficiency as $[KP]$ grows with the number of problems successfully addressed. By building such a system, we have achieved the following contributions in this work.

- 1) A system that is capable of continual learning and solving complex Boolean problems. The tested problems are considered intractable to isolated learning.
- 2) Methods to automatically form a learning curriculum when presented with multiple problems at once. The system determines the next problem that is most appropriate to address, and thereby removes the requirement of human intervention, e.g., in LL [17], [26].
- 3) The representation of solutions to enable a clear understanding of learned problems, cf. eXplainable AI [27]. This interpretability of the learned solutions also provides knowledge dependencies among problems.

There is an important paradigm shift in designing test problems when switching from learning systems that consider a single problem to continual learning systems. As a human cannot learn integration without first knowing addition, the building blocks (both knowledge and skills to manipulate the knowledge) must be made available. In genetic programming (GP), the researcher defines the function and terminal set, while, in ConCS, it is the problems themselves and the skills/functions they provide that are paramount, e.g., once the Boolean “AND” problem is learned, the “AND” function can be reused along with its component knowledge.

ConCS will be tested with multiple problems, in this case 19, including regression and classification problems, to demonstrate its learning capabilities. A range of well-known complex Boolean problems was selected, e.g., Mux, Parity, Majority-on, and Carry problems. These problems can be used to form hierarchical problems, i.e., where the solutions of one type of problem can form the input to another,

³The input arguments of the function are equivalent to the conditions of the rules, where the resultant action is the function’s returned value.

⁴Curriculum: Orders of tasks depict the connection of knowledge amongst tasks and their dependencies.

such that they exhibit a measurable search space, dependency structure, multimodal solutions, and distributed niches (sub-solutions) [28]. Importantly, the feature space is not linearly separable as the problems exhibit heterogeneity (the same class can be caused by different input modes) and epistasis (the importance of one feature depends on other feature values). Unlike many real-world problems they have precisely known solutions such that the correctness of the produced solution can be evaluated.

As well as the positive reasons for using Boolean problems as the test suite, there were reasons not to immediately use real-world problems—although CFs can accept real-world (integer, float, symbolic, categorical, and so forth) features that can be processed in GP-tree like structures. First, the signal-to-noise ratio is often unquantified, so even with the error threshold parameter enhancing noise reduction in LCSs, it would be impossible to disambiguate poor performance caused by poor algorithm performance versus poor signals related to the patterns. Second, and most importantly, the datasets are not yet curated for continuous learning, i.e., the data sets are produced at too high a level of complexity without the underlying problems being available that teach the necessary building blocks. However, ConCS is designed to be readily adaptable for future implementation in real-world/real-valued domains.

II. BACKGROUND AND RELATED WORK

A. Learning Classifier Systems

LCSs refer to a family of rule-based online-learning algorithms in the fields of EC and machine learning adopted from the study of cognition [23], [24], [28]. Unlike many other EC algorithms, LCSs have been used directly to solve machine learning and robotic tasks instead of addressing optimization tasks. A characteristic strength of LCSs is the capability of dividing a problem into smaller niches (subproblems) where each can be resolved by a subset of its solution [24]. The niche of a rule becomes larger when the rule becomes more generalized by the pressure toward generality within LCSs. Among LCSs, CCS [29], and CXCS [30] were the first LCSs that managed to link knowledge among rules, so can be considered *agents*.

XCS is an online reinforcement learning LCS adapted for machine learning and robotics problems, especially classification tasks and maze problems [13], [31]. An XCS evolves its population of rules interacting with an environment representing the target problem. Each rule is in the form of “if *condition* then *action*,” where the *condition* tells whether the rule matches the perceived environment state, and the *action* part specifies the action to be executed on the environment. The rule conditions enable divide-and-conquer by matching the rule with an environment niche. Here, a rule can compete with other rules in the niche to be chosen for performing its action. A rule is also known as a classifier when incorporating its performance statistics. XCS for Boolean problems traditionally represents the actions of rules by numeric constants and the conditions by ternary alphabet $\{0, 1, \#\}$. By correlating the fitness of a rule with the prediction accuracy of the rule, XCSs

can form a complete map from pairs of (state, action) to its estimated returns.

The original ternary alphabet in XCS performs well if an optimal parameter configuration can be determined [32]. However, it is still limited in scalability as generalization is only provided through the “don’t care” ($\#$) symbols [9], which remove redundant/irrelevant features. In order to better partition the search space, i.e., not just in hyper-rectangles, alternative encodings have been proposed, e.g., neural networks [33], [34], [35], [36], [37], [38], tree-based programs [9], [39], [40], [41], [42], [43], and cyclic graphs [44].

B. Scalable LCSs

In order to scale to complex problems, CFs [9] were developed to encapsulate blocks of learned knowledge to transfer from small to more complex problems in a single domain. CFs are small GP-like tree-based programs with the depth initially limited to 2, so this representation is flexible and highly interpretable. There have been many advancements to use CFs to improve learning scalability of XCS. Iqbal et al. [9], who first introduced the concept of CFs, used them in the rule conditions, cf. XCSCFC. Their experiments demonstrated the scalability of XCSCFC with transfer learning by learning Mux problems from small scales to large scales. XCSCFC was the first XCS that successfully solved the 135-bit Mux problem. However, using this method to solve hierarchical problems, where an overarching problem consists of subproblems, was still an unanswered question. Such problems are difficult for machine learning algorithms as the search space consists of multiple interacting patterns that are often repeated, which obfuscates the decision boundaries. Based on XCSCFC, Alvarez et al. [16] later introduced ruleset functions in CFs in XCSCF² to enable the reusability of CFs with learned ruleset functions for the function nodes. This system can solve hierarchical problems, such as 18-bit Hierarchical Mux problem, with transfer learning. The scaling of building blocks represented by CFs is in the bottom-up manner, which grows from elemental building blocks to complex ones. This is opposite to Automatically Defined Functions [45], a tree-based program that enables learning functions in tree nodes. These works achieved high performance but required a customized sequence of LL with manual transferring criteria.

Several works later improved the autonomy of CF-based XCSs. CF conditions in XCS were extended with an extra population of CFs to address the problem regarding the large search space in tree-based programs in XOF [46], [47]. XOF used a new parameter CF-fitness and an observed list of preferable CFs to improve the learning performance and enhance the generalization of rule conditions. XOF could solve complex problems with no upper bound for the depth of CFs. Nguyen et al. [48] introduced mXOF with a new relatedness parameter as a relationship measure (magnitude) among tasks in multitask learning (MTL) system to transfer features efficiently without crafting transferring criteria. Also, the ability to automatically adjust feature transferring is particularly appropriate for systems that grow tree-based features as the set of features for each learning system changes during the

learning process. mXOF improved the learning performance when multiple tasks were supportive and also reduced negative transfer in the case of unrelated tasks.

Another approach to increasing the generality of XCS rules is to use CFs in rule actions. Rule actions using CFs can flexibly adapt action values regarding the environment state. The advent of this approach was XCSCFA [43] that has the ability to achieve high-level knowledge. Alvarez et al. [17], [26] later extended the reusability of ruleset functions in XCSCFA to produce XCSCF*. XCSCF* can transfer ruleset functions between XCSCFA systems where each addresses a problem in a sequence. The sequence was ordered in a curriculum to guarantee the increasing level of complexity and compositionality, i.e., LL [49]. This system was able to find a general solution for the Mux problem domain, which can solve the problem at any scale. CFs were later used to extend the capability of XCS in multistep problems [50]. The CFs in XCS rules enable XCS to overcome the aliasing problem and thereby solve complex maze problems.

However, all these attempts to integrate the ability to reuse knowledge required human intervention, such as specifying the numbers of transferred CFs (XCSCFC) and designing a curriculum (XCSCF² and XCSCF*).

C. Other Related Work

This work develops a complex classification system that uses transfer learning to learn in both multitask (parallel) learning and continual learning with incremental progress. In the field of LCSs, this is the first system with such properties. However, there have been related work in other AI fields with analogous abilities [51], [52], [53], [54]. In the EC field, Da et al. [51] introduced an EC framework that enables online learning and exploitation of similarities across multiple *optimization* problems. A salient aspect of the framework is that it accounts for latent similarities that are not apparent on the surface but may be revealed during the evolutionary search. Other than EC, Tommasino et al. [52] developed a transfer expert reinforcement-learning model that acquires multiple skills, through the skill-to-skill knowledge transfer, to enable learning processes to focus on the novel aspects of the new skills. A key feature of this model is the capacity of its gating networks to accumulate, in parallel, evidence on the capacity of experts to solve the new tasks to increase the action responsibility. Next, lifelong metric learning is another online learning approach to mimic “human learning” in using previous experiences to help learn new tasks [53]. This new metric learning framework maintains a common subspace for all learned metrics, transfers knowledge from the subspace to learn each new task, and redefines this subspace over time to maximize performance across all metric tasks. AutoML-Zero was designed to build machine learning algorithms from scratch using evolutionary methods [54]. This was aimed to reduce human bias in the algorithm design and indirectly enable more autonomy to designed AI systems. AutoML-Zero constructs an algorithm by filling three elementary functions, i.e., *Setup*, *Predict*, and *Learn*, but this requires a prerequisite set of basic arithmetic operations.

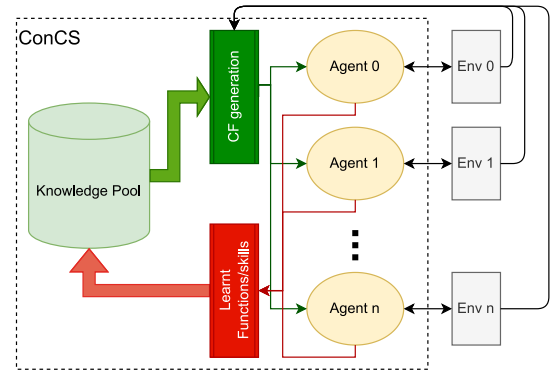


Fig. 1. ConCS model. Each environment (Env) represents a task.

One may relate the learning paradigm of ConCS with MTL [55]. The general concept of MTL is to learn multiple different tasks together to improve the learning performance of each task. Technically, the proposed ConCS also learns multiple tasks together, but it differs from standard evolutionary approaches to MTL where the objective is optimization (MTO) rather than classification [10]. ConCS not only addresses increasingly more complex tasks (the complexity of tasks is set *a priori*, and remains fixed in standard MTO) but also serves as a general Boolean problem-solving system. This includes being able to handle unrelatedness among the presented tasks. Furthermore, a general problem-solving system should also be able to solve problems arriving at any arbitrary time, unlike standard MTO where tasks (typically only two) are presented simultaneously. For these reasons, the learning paradigm of ConCS incorporates core aspects of LL, continual learning [5], and MTL, where the system learns multiple problems/tasks incrementally and continually.

III. CONCS: CONTINUAL CLASSIFIER SYSTEM

The ConCS is composed of multiple agents where each is dedicated to a task (see Fig. 1 and Algorithm 1), i.e., solving a specific problem. The target of the whole ConCS is to accumulate knowledge from its agents. This global target is expected to support the problem-solving capability of the agents of ConCS. Note the global target is not needed *a priori*, plus a new target can be given once an old one has been reached without the need for retraining. The proposed system spawns a new agent for each task. The objectives of each agent are not only solving its task, but also accumulating knowledge from solving its task into the knowledge pool of ConCS, which is the common goal of the whole ConCS. The problem-solving capability is validated with its ability to scale and comprehend harder tasks with increasing complexity.

Communication among agents in ConCS is indirect and limited to interactions through the knowledge pool. The knowledge pool consists of learned populations (can be reused as functions), CFs (can be reused with or without the function where they were originally generated), and axiomatic functions/skills/terminals along with the input/output type of each of these elements. An agent can extract skills and functions from the knowledge pool to reuse after it has learned a

Algorithm 1 Workflow of Coordinating Agents and Accumulating Knowledge

```

1: Start all agents
2: Total progress  $P_{total} = \sum_{\text{all agents } i} P_{0,i}$ , where  $P_{0,i}$  is the default progress of agent  $i$ 
3: while exist any unsolved problems do
4:   Roulette wheel to run the next iteration of an agent  $A$  regarding agents' progress
5:   if Agent  $A$  solves its problem then
6:     if first time then
7:       Compact, Add its ruleset to the knowledge pool
8:     if Current ruleset of agent  $A$  differs from the rulesets of existing knowledge then
9:       Compact, Add its ruleset to the knowledge pool

```

problem, it can append its completed function/skill to the pool. That is, a function (skill) maps from input to output (input to procedure), which can be learned through rules mapping conditions (subsets of input) to actions (output).

Continual MTL produces challenges for ConCS. First, ConCS without the curricula might pick up a complex task by chance and get stuck there. The intuitive resolution is that ConCS should focus on the easiest task, identified by most often having positive environmental feedback with the least effort. This leads to a need to prioritize agents for access to the CPU. Ultimately, ConCS will be configured for truly parallel GPU-based cloud computing but initial development is kept simple to avoid confounding factors. Second, the search space becomes larger as more knowledge is available. To add limits to the search space, the type-fitting XCSCFA introduced in XCSCF* [26] is also used as the learning agent in ConCS to learn both subproblems and target problems. XCSCFA with the type-fitting property can divide the search space into smaller sections by compatible types [26]. This enables accumulating more knowledge without making the search space of each agent intractable. Therefore, we propose two key components of ConCS to address these issues: 1) type-fitting XCSCFA and 2) knowledge management. These components will be described in the following sections. We will also provide a brief explanation of all subproblems designed to bootstrap the system's learning performance.

A. Type-Fitting XCSCFA

We employed type-fitting XCSCFA [26] as the algorithm for the agents of ConCS. This version of XCSCFA generates verifiable CFs (typed CFs) as it enables the type-fitting property in generated CFs. This property guarantees connected nodes within generated CFs to be compatible with one another and the CFs' input and output to be compatible with the problem (environment). Although ConCS is not limited to using only XCSCFA, this algorithm is suitable to learn the high-level logic behind the tested problems as it can address both regression and classification problems. ConCS must be able to spawn an agent with a suitable algorithm (or multiple agents) if no prior experience regarding a new problem is detected.

Generating Typed CFs (T-CFs) applies a top-down recursive process of generating tree nodes, i.e., the function *genNode* illustrated in Algorithm 2. In ConCS, we keep the depth limit of base CFs as 2, as is the original definition of CFs [9]. Generating

Algorithm 2 Typed CFs Are Generated Based on a Recursive Function for Generating Nodes. The Function Is Given the Set of Action Types T_a , the Type Set of Base CFs T_b , the Expected Output Types T_o , the Expected Input Types T_i , the Intermediate Level l_i , and a Clustered Set of All Functions S_f

```

1: procedure GENNODE( $T_o, T_i, l_i$ )
2:    $T'_i = \emptyset$  (empty set)
3:   if  $l_i = 2$  then
4:     Output types  $T_o = T_a$ 
5:   if  $l_i = 1$  then
6:     Output types  $T'_i = T_b \cup \{integer\}$ 
7:   Filter function set  $S_{filtered}$  from  $S_f$  by required output types  $T_o$  and input types  $T_i$ 
8:   Function  $f = \text{randomSelect}(S_{filtered})$ 
9:   for index  $i$  in  $f.inputs$  do
10:    if  $l_i - f.level > 0$  and  $\text{random}[0, 1) < 0.5$  then
11:       $f.inputs[i] = \text{GENNODE}(f.input\_types[i], T'_i, l_i - f.level)$ 
12:    else
13:      Set of compatible base CFs  $S_{bCF} = \emptyset$ 
14:      if  $integer \in \text{function.input\_types}[i]$  then
15:         $c = \text{randomSelect}([1, ..., \text{len}(Atts)])$ 
16:      for  $cf_{base}$  in all base CFs do
17:        if  $cf_{base}.out\_types \& f.input\_types[i] \neq \emptyset$  then
18:          Add  $cf_{base}$  to  $S_{bCF}$ 
19:       $f.inputs[i] = \text{randomSelect}(S_{bCF})$ 

```

a new CF needs to match with the action types of the problem and available output types from the base CFs (CFs that encode data input), i.e., data input types. First, the top node of a T-CF must employ a function with output types compatible with the action types of the problem. Then, the process recursively builds lower-level nodes that satisfy the type-fitting property. At any point when generating nodes, there is also a fixed probability of 0.5 for generating a leaf node from base CFs, which stops the CF from going any deeper.

There are four possible types in this implementation: 1) Boolean; 2) integer; 3) float (real numbers); and 4) list. While Boolean variables are compatible with integers and floats, and integers are compatible with floats, the compatibility does not follow the opposite way. Lists are not compatible with other types. Other details about the type-fitting property of XCSCFA with the type-fitting property can be found in [26]. We also propose several improvements for XCSCFA [43] to process redundant genotypes of functions (function versions). The following section will describe CF-equality checking, which is related to function genotypes.

CF Equality: Checking equality between CFs in ConCS is needed when adding new rules to the rule population, adding new CFs to the CF population, function compaction (in each agent, described later in Section III-C), and adding a function to the knowledge pool to prevent unnecessary duplication that slows computation and can obscure knowledge. Checking CF equality is not as simple as checking the equality of all their corresponding nodes (Algorithm 3). Two CFs are generally equal if they have the same genotype. However, because there are distinct versions of the same functions, the criterion of genotypic equality varies case by case. A genotypic difference in the reused function makes two CFs unequal in general learning processes of each problem. This enables diversity by function genotypes. This diversity is rational because two genotypes of a function can behave unevenly in problems

Algorithm 3 Recursive Process of Comparing Two CFs. The Flag $f_version$ Determines Whether the Difference of Functions by Versions Only Makes the CFs Different, i.e., Function Returns False

```

1: procedure COMPARECFs( $cf_1, cf_2, \text{flag } f\_version$ )
2:   if ( $cf_1.function \neq cf_2.function$ )  $\vee$  ( $f\_version \wedge (cf_1.f.version \neq$ 
    $cf_2.f.version)$ ) then
3:     return False
4:   if  $\text{len}(cf_1.inputs) \neq \text{len}(cf_2.inputs)$  then
5:     return False
6:   for  $\text{input } i_1 \in cf_1.inputs \wedge i_2 \in cf_2.inputs$  do
7:     if  $i_1, i_2$  are CFs then
8:       if  $\neg \text{COMPARECFs}(i_1, i_2, f\_version)$  then
9:         return False
10:    else
11:      if  $i_1 \neq i_2$  then
12:        return False
13:  return True

```

other than the one that produces the function. On the contrary, the inequality caused by function genotypes is ignored when checking the equality of classifiers during compacting solutions (see Section III-C). Even though they have different versions of functions f , which might create unexpected behavioral differences, they do not demonstrate any distinction of knowledge.

B. Stochastic Task Preference

Selecting the next task to be run has a minor contribution to ConCS as it can still solve the problems with a naïve uniform distribution of task prioritization. We designed a simple heuristic method to prioritize agents/tasks with higher learning progress. This is to reduce the total computation time by focusing on tasks with the high potential of being solvable. Therefore, repeated Roulette Wheel selections based on agents' learning progress determines which agent to run at each iteration until there are no more operating agents. In this work, we define the learning progress as a parameter correlated with the absolute accuracy and the improvement of the accuracy of the agent

$$\text{progress} = \max(0.1 * \text{accuracy}_{\text{adj}}, \Delta \text{accuracy}) \quad (1)$$

where $\text{accuracy}_{\text{adj}}$ is normalized to initially start at approximately 0.5 for all agents and 1.0 when the problem is solved. This leads to an initial progress value of approximately 0.05 when there is no increase in accuracy. The normalization is as follows:

$$\text{accuracy}_{\text{adj}} = \begin{cases} \frac{\text{accuracy} * |\text{actions}| - 1}{|\text{actions}| - 1} * 0.5 + 0.5 & \text{if classification} \\ \text{accuracy} / 2 + 0.5, & \text{if regression.} \end{cases} \quad (2)$$

It can be inferred that the frequency of updating agent progress is equal to the frequency of updating agent accuracy, which is set to once every 500 iterations (250 explored instances) for all agents.

C. Knowledge Management

The knowledge pool is the collection of built-in and obtained skills/functions/CFs. In other words, it is the function set listing all available knowledge (how features through

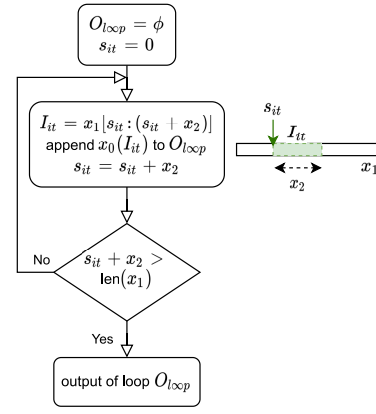


Fig. 2. Flowchart of a function with the loop skill. This function iteratively applies another function x_0 on x_1 with a step size of x_2 and concatenates the iterative output in O_{loop} .

functions/skills are related to higher order features (CFs) and ultimately to actions). Agents search for solutions by combining functions from this function set and their base CFs. At the beginning, the knowledge pool has prerequisite knowledge, termed built-in axioms (see Table I). These built-in axioms must include the building blocks required to construct solutions for the target problems (see Section III-D), which is common practice in GP algorithms [56]. In addition to the necessary building blocks, the knowledge pool also provides general functions for Boolean problems that might or might not be useful, with the assumption that ConCS should be able to choose the appropriate functions. While the majority of these functions are general knowledge that can be reused in many other problems, some of them are tailored for these problems and may not be generally applied.

The *general loop* is a general skill that requires a core process, i.e., a function, given by the input x_0 to become a function. *General loop* iteratively applies function x_0 on input list x_1 with a moving starting point s_{it} (Fig. 2). In the first iteration, function x_0 processes x_1 from the first item. At each iteration, the starting point on x_1 moves x_2 steps from the preceding iteration to extract the input for x_0 . The loop ends when the starting point moves beyond the end of x_1 . The output of a *loop* is the concatenated list of outputs of x_0 in all its iterations.

We do not divide the initial subproblems any further although it was demonstrated that CF-based XCSs could learn such functions from even smaller subproblems [16]. Future work with ConCS will explore the intellectually interesting question of “what are smallest axioms that can initialize learning?”

Function Compaction: When an agent solves a problem successfully for more than 500 instances, it will try to compact its population to extract a solution, as a function, to its problem. The steps are listed in Algorithm 4. First, the agent selects only experienced ($\text{exp} \geq \theta_0$), max-prediction ($P = 1000$ commonly), and accurate classifiers ($\text{err} = 0$) from its population. Then, it finds the highest fitness f_{max} in its classifier population. All classifiers having low fitness (i.e., $f < 0.5 * f_{\text{max}}$) are filtered out. Next, the agent tries to subsume all over-specific classifiers.

TABLE I
AXIOMATIC SKILLS AND FUNCTIONS. THE OPERATOR $x[a, b]$ IS TO EXTRACT A LIST OF ITEMS OF THE LIST x STARTING FROM THE INDEX a TO, BUT NOT INCLUDING, THE END INDEX b . BINARY NUMBERS HAVE TWO VALUES (0, 1).
 $\oplus$ AND $\ominus$ ARE BINARY ADDITION AND SUBTRACTION, RESPECTIVELY

Skills & Functions (tag)	Input Length (inputs)	Input types	Output type	Operation
AT (@)	2 (x_0, x_1)	list, integer	any of Boolean, integer, float, list	$x_0[x_1]$
LENGTH (<i>len</i>)	1 (x_0)	list	integer	$len(x_0)$
AND ($\wedge$)	2 (x_0, x_1)	Boolean	Boolean	$x_0 \wedge x_1$
OR ($\vee$)	2 (x_0, x_1)	Boolean	Boolean	$x_0 \vee x_1$
XOR ($\otimes$)	2 (x_0, x_1)	Boolean	Boolean	$x_0 \otimes x_1$
NOT ($\neg$)	1 (x_0)	Boolean	Boolean	$\neg x_0$
FLOOR (<i>floor</i>)	1 (x_0)	integer, float	integer	$\lfloor x_0 \rfloor$
CEIL (<i>ceil</i>)	1 (x_0)	integer, float	integer	$\lceil x_0 \rceil = \lfloor x_0 + 1 \rfloor$
SUMMATION of list items (<i>sum</i>)	1 (x_0)	list	integer, float	$sum(x_0)$
ADD (+)	2 (x_0, x_1)	integer, float	integer, float	$x_0 + x_1$
SUBTRACT (<i>subt</i>)	2 (x_0, x_1)	integer, float	integer, float	$x_0 - x_1$
MULTIPLY (<i>mul</i>)	2 (x_0, x_1)	integer, float	integer, float	$x_0 * x_1$
DIVIDE (<i>div</i>)	2 (x_0, x_1)	integer, float	integer, float	x_0 / x_1
GREATER (<i>isGreater</i>)	2 (x_0, x_1)	integer, float	Boolean	$x_0 > x_1?$
HEAD (<i>head</i>)	2 (x_0, x_1)	list, integer	list	$x_0[0 : x_1]$
TAIL (<i>tail</i>)	2 (x_0, x_1)	list (length l), integer	list	$x_0[x_1 : l]$
GENERAL LOOP (<i>loop</i>)	2 (x_0, x_1, x_2)	function, list, integer	list	convolute x_1 by function x_0 with problems size x_2
LOG2 (<i>log2</i>)	1 (x_0)	integer, float	integer, float	$log_2(x_0)$
BINADD (<i>binadd</i>)	2 (x_0, x_1)	list (of Boolean)	list (of Boolean)	$x_0 \oplus x_1$
BINSUB (<i>binsub</i>)	2 (x_0, x_1)	list (of Boolean)	list (of Boolean)	$x_0 \ominus x_1$
BIN2DEC (<i>bin2dec</i>)	1 (x_0)	list (of Boolean)	integer	convert Binary number to Decimal number
MODULO (<i>mod</i>)	1 (x_0)	integer	integer	$x_0 \% 2$
CONSTANT (<i>c</i>)	0 ($\emptyset$)	N/A	integer	return a constant number

Algorithm 4 Function Compaction When a Problem Is Solved

- 1: Make a copy of the classifier population
- 2: Select only accurate and experienced classifiers
- 3: Get max classifier fitness from the population $f_{\max}^{cl}$
- 4: Remove classifiers with fitness $< 0.5f_{\max}^{cl}$
- 5: Compare and remove duplicate classifiers (action CFs with the same structure but different function versions are also duplicate)
- 6: Remove classifiers that can be subsumed by any other classifier remaining in the population
- 7: **if** the remaining population solves the problem **then**
- 8: Add to the knowledge pool

ConCS needs to check the equality of ruleset functions to avoid adding functions with the same logic more than once to the knowledge pool, which undesirably enlarges the search space of all agents. In this case, functions are considered equal if they contain the same rule sets. ConCS confirms the equality of two ruleset functions by matching all rules of the ruleset of one function to all rules in the rule sets of other ruleset functions. In function compaction, the equality of CF actions ignores the differences of function genotypes (see Section III-A for the justification).

D. Target Problems

Although real-world domains contain reusable patterns, existing benchmark datasets are often separate in the patterns they contain, e.g., UCI Zoo, Wisconsin Breast Cancer, and Sonar datasets, where a discovered sample distribution is not present in another dataset. We need a set of problems with feature patterns that are constructed from subpatterns to

test the scaling capability of ConCS in related problems. We also need separate problems as continual learning might face independent problems with distinct patterns. Boolean domains satisfy these criteria because they have known solutions and are interrogable. Therefore, target problems are four hierarchical problems and 15 subproblems to be solved continually and in parallel with the hierarchical problems. These subproblems provide knowledge that is potentially prerequisite for the target problems. In the Mux domain, most of the subproblems are identical to the subproblems used in XCSCF* [26].

We designed the learning stages of the target problems and subproblems with variable scales, i.e., lengths of input bits, to require successful solutions, if any, to be scale invariant. Our experiments show that when learning fixed-scale problems, constants (CFs) [26] can contribute to generating solutions that are only valid at the fixed scale, which inhibits scaling. Being scale invariant means that successful solutions are capable of solving these subproblems at any scale.

The rationale behind the selection of the 19 Boolean problems (see Table II for an overview and Supplementary material for a complete description) was threefold: 1) the most difficult hierarchical problems that have been widely tested in the LCS literature should be included; 2) likely problems with functionality that could be useful in addressing 1) should also be included; and 3) the problems should not be overly curated to 1) and 2) such that surprising curricula might arise. Finally, past research has shown that LCS variants, even with proper parameter tuning, fail to learn as the search space scales [32] due to having to learn decision boundaries from scratch as either the search space or the number of possible decision

TABLE II
TO BE LEARNED FUNCTIONS/SKILLS. A FEW OF THEM ARE HIGHLY REUSABLE IN GENERAL CASES. $len()$ IS A FUNCTION TO RETURN THE LENGTH OF THE INPUT WITH TYPE *list*. $(x_0, x_1, \dots)$ ARE INPUTS OF THESE FUNCTIONS

Id	Functions (abbreviations)	Inputs	Input types	Output type	Anticipated operation to be learnt
0	Address Length given Mux size	x_0	integer	integer	$\lfloor \log_2(x_0) \rfloor$
1	Address Length given Mux bitstring	x_0	list	integer	$\lfloor \log_2(len(x_0)) \rfloor$
2	Address Bits	x_0	list	list	$x_0[0 : \lfloor \log_2(len(x_0)) \rfloor]$
3	Decimal value of Address Bits	x_0	list	integer	$bin2dec(x_0[0 : \lfloor \log_2(len(x_0)) \rfloor])$
4	Data Bit Position	x_0	list	integer	$\lfloor \log_2(len(x_0)) \rfloor + bin2dec(x_0[0 : \lfloor \log_2(len(x_0)) \rfloor])$
5	General Multiplexer (<i>mux</i>)	x_0	list	Boolean	$x_0[\lfloor \log_2(len(x_0)) \rfloor + bin2dec(x_0[0 : \lfloor \log_2(len(x_0)) \rfloor])]$
6	Half String Size	x_0	list	integer	$len(x_0)/2$
7	First Half	x_0	list	list	$x_0[0 : (len(x_0)/2)]$
8	Second Half	x_0	list (length l)	list	$x_0[(len(x_0)/2) : l]$
9	Binary Addition of 2 halves	x_0	list	list	$x_0[0 : (len(x_0)/2)] \oplus x_0[(len(x_0)/2) : l]$
10	Length of Binary Addition	x_0	list	integer	$len(x_0[0 : (len(x_0)/2)] \oplus x_0[(len(x_0)/2) : l])$
11	General Carry-one (<i>carr</i>)	x_0	list	Boolean	$len(x_0[0 : (len(x_0)/2)] \oplus x_0[(len(x_0)/2) : l]) > len(x_0)/2$
12	Sum Modulo 2	x_0	list	Boolean	$sum(x_0) \% 2$
13	General Even-parity (<i>epar</i>)	x_0	list	Boolean	$sum(x_0) \% 2 = 0?$
14	General Majority-on (<i>maj</i>)	x_0	list	Boolean	$sum(x_0) > len(x_0)/2?$
15	Hierarchical Multiplexer	x_0	list	Boolean	$mux(X)$ where $X = loop(epar, x_0, 3)$
16	Hierarchical Carry-one	x_0	list	Boolean	$carr(X)$ where $X = loop(epar, x_0, 3)$
17	Hierarchical Majority-on	x_0	list	Boolean	$maj(X)$ where $X = loop(epar, x_0, 3)$
18	Hierarchical Even-parity	x_0	list	Boolean	$epar(X)$ where $X = loop(epar, x_0, 3)$

boundaries is too large to begin to find and recombine useful blocks of knowledge.

IV. EXPERIMENTS

We evaluated the proposed ConCS by solving 19 different problems. Each agent is a type-fitting XCSCFA with a common configuration. We examined whether the system (without human-guided customization) can work on all problems. The settings for all agents are the general configuration for XCS [31]: the population size of 1000; the learning rate $\beta = 0.2$; the probability of crossover operation in reproduction $\chi = 0.8$; the probability of mutation $\mu = 0.04$; the probability of using a don't care in each classifier condition when covering $P_{don'tCare} = 0.33$; the experience threshold for a classifier to be a subsumer $\theta_{sub} = 20$; the initial fitness value assigned to new classifiers $F_o = 0.01$; the fraction of classifiers selected for tournament selection from an action set 0.4; error threshold $\epsilon_0 = 10.0$. Additionally, the minimum number of actions in the match set θ_{mna} was set to 4 for both classification and regression problems. This value of θ_{mna} encourages each agent to create more genotypes to increase the chance of obtaining the desired CF action.

Each experiment was run 30 times with 30 fixed random seeds. The stopping criteria were when the agent consistently maintained 100% accuracy for at least 50 000 instances, or when it reached the maximum learning instances for each agent, 2 000 000 instances, which was chosen to be arbitrarily large.

The first experiment of ConCS was run on 19 problems in parallel with 19 agents starting at the same time to show the discovery of the network of knowledge (corresponding to 19 problems). Then, the performances of ConCS were compared with those of XCSCF* using type-fitting XCSCFA, on different sets of problems. In these experiments, XCSCF* shared the same configuration and the problem sets with agents of

ConCS, except that XCSCF* runs sequentially. ConCS and XCSCF* are also compared with XCS and XCSCFC in solving target problems at specific scales. XCS and XCSCFC are configured with their empirical configurations, which require much larger population sizes. Solutions yielded by these two approaches are limited to solving the tested problems at fixed scales. Additionally, another comparison of ConCS with XCSCF* and popular machine learning algorithms in terms of accuracy in supervised learning was executed to assess the generalization of ConCS' solutions.

Finally, an extra experiment with random arrivals of the 19 problems was performed to show the ability of ConCS to learn continually and learn multiple tasks in parallel. A starting problem was chosen randomly. The other 18 problems were initialized at random times from the starting point using a uniform random generator within the range of 0 to 1 h. The arrival times were generated once and fixed in all 30 runs.

A. Discovered Knowledge

Experiments on all 19 problems demonstrated the ability of ConCS to achieve 100% performances on all problems in all 30 runs (see Section IV-B for details on learning performances). Table III shows the learned solutions from the agents. These acquired solutions are interpreted from the actions of the rules in compacted solutions. It is noted that rule conditions of all these acquired solutions are composed of only *don't care*, which is expected as the learned CF in the rule action addresses all possible inputs correctly.

For several problems, such as the Even-parity problem, solutions acquired for the same problem involve distinct genotypes in the CF action, where some contain bloat or inefficient solutions. However, the bloat in such solutions is limited. Thus, it is trivial to verify that diverse solutions for each problem are equal. Also, acquired solutions are highly interpretable. Thus, it is straightforward to confirm that these solutions, shown in

TABLE III
LEARNED FUNCTIONS/SKILLS. *attlst* IS A BASED CF REPRESENTING THE LIST OF ALL INPUT ATTRIBUTES. *i* AND *k* REFER TO ARBITRARY INTEGERS IN CONSTANTS (CFs)

Skills & Functions	Inputs (based CFs)	Function Name	Learned Solutions
Address Length given Multiplexer size	<i>attlst</i> , x_0	<i>mux_addr_length</i>	$\text{floor}(\log_2(x_0))$
Address Length given MUX attributes	<i>attlst</i> , $x_0, x_1, \dots$	<i>mux_addr_length2</i>	$\text{mux_addr_length}(\text{len}(\text{attlst}))$
Address Bits	<i>attlst</i> , $x_0, x_1, \dots$	<i>mux_addrbits</i>	$\text{head}(\text{attlst}, \text{mux_addr_length2}(\text{attlst}))$
Decimal value of Address Bits	<i>attlst</i> , $x_0, x_1, \dots$	<i>mux_gate</i>	$\text{bin2dec}(\text{mux_addrbits}(\text{attlst}))$
Data Bit Position	<i>attlst</i> , $x_0, x_1, \dots$	<i>mux_databit</i>	$\text{add}(\text{mux_gate}(\text{attlst}), \text{mux_addr_length2}(\text{attlst}))$
Variable-size Multiplexer	<i>attlst</i> , $x_0, x_1, \dots$	<i>mux</i>	$\text{@}(\text{attlst}, \text{mux_databit}(\text{attlst}))$
Hierarchical Multiplexer	<i>attlst</i> , $x_0, x_1, \dots$	<i>hpar</i>	$\text{mux}(\text{loop}(\text{epar}, x_0, 3))$
Sum Modulo 2	<i>attlst</i> , $x_0, x_1, \dots$	<i>epar_remainder_2</i>	$\text{mod}(\text{sum}(\text{attlst}), 2)$
Variable-size Even-parity	<i>attlst</i> , $x_0, x_1, \dots$	<i>epar</i>	$\neg(\text{epar_remainder_2}(\text{attlst}))$ $\text{greater}(c(1), \text{epar_remainder_2}(\text{attlst}))$ $\text{greater}(\text{div}(i, k), \text{epar_remainder_2}(\text{attlst})) \ (i \leq k)$
Hierarchical Even-parity	<i>attlst</i> , $x_0, x_1, \dots$	<i>hpar</i>	$\text{epar}(\text{loop}(\text{epar}, x_0, 3))$ $\text{epar}(\text{loop}(\text{epar}, x_0, 1))$
Half String Size	<i>attlst</i> , $x_0, x_1, \dots$	<i>half_length</i>	$\text{div}(\text{len}(\text{attlst}), 2)$ $\text{mul}(\text{len}(\text{attlst}), \text{div}(c(i), c(2i)))$
First Half	<i>attlst</i> , $x_0, x_1, \dots$	<i>car_headstring</i>	$\text{head}(\text{attlst}, \text{half_length}(\text{attlst}))$
Second Half	<i>attlst</i> , $x_0, x_1, \dots$	<i>car_tailstring</i>	$\text{tail}(\text{attlst}, \text{half_length}(\text{attlst}))$
Binary Addition of 2 halves	<i>attlst</i> , $x_0, x_1, \dots$	<i>car_binadd</i>	$\text{binadd}(\text{car_headstring}(\text{attlst}), \text{car_tailstring}(\text{attlst}))$
Length of Binary Sum	<i>attlst</i> , $x_0, x_1, \dots$	<i>car_lenbinadd</i>	$\text{len}(\text{car_binadd}(\text{attlst}))$
Variable-size Carry-one	<i>attlst</i> , $x_0, x_1, \dots$	<i>carr</i>	$\text{greater}(\text{car_lenbinadd}(\text{attlst}), \text{half_length}(\text{attlst}))$
Hierarchical Carry-one	<i>attlst</i> , $x_0, x_1, \dots$	<i>hcar</i>	$\text{carr}(\text{loop}(\text{epar}, x_0, 3))$
Variable-size Majority-on	<i>attlst</i> , $x_0, x_1, \dots$	<i>maj</i>	$\text{greater}(\text{sum}(x_0), \text{half_length}(\text{attlst}))$
Hierarchical Majority-on	<i>attlst</i> , $x_0, x_1, \dots$	<i>hmaj</i>	$\text{maj}(\text{loop}(\text{epar}, x_0, 3))$

Table III, are identical to the logic of the 19 given problems in Table II. However, in certain cases, the evolved solutions deliver new and unexpected insights into the problems (see Supplementary material).

The final solutions from Table III on all problems form a network of knowledge, where we can learn about the dependencies of one problem on others. Fig. 3 illustrates the learned network of knowledge. An arrow directed from a problem A or a preprovided function f to a problem B means the solution for B uses the solution (i.e., learned function) discovered from learning problem A or function f . ConCS found that the *Half String problem* is one of the most generally reused as it is used in at least four other problems (*car_headstring*, *car_tailstring*, *carr*, *maj*). For the innately provided skills, general loop loop, constant function c , and length len are the three most commonly used functions. The constant function c is the most used one as it creates the base CF *attlst* for almost all solutions. On the contrary, others, such as binary operators $[\wedge, \vee, x]$ and binary subtraction, were found redundant as they were never used in any learned solution.

New Understanding of Hierarchical Even-Parity Problem:

Table III lists all discovered solutions for the Hierarchical Even-parity *hpar* problem. In addition to the expected first rule on the Table, ConCS also yields the second unanticipated rule in all 30 runs. This rule proposes a new understanding of the Hierarchical Even-parity problem that was unexpected before experiments. Specifically, the lower level of the Even-parity loop with step 1 (represented by CF $c(1)$), is the Even-parity problem on each bit of the input. The Even-parity problem on one-bit inputs can be interpreted as the negation of the

only bit in the input. Therefore, the second rule proposes that the Hierarchical Even-parity problem is equivalent to a “flat” Even-parity problem on the bitwise negation of the input bitstring. This finding is validated in the Supplementary material.

B. Learning Performances

Fig. 4 illustrates the learning performances of ConCS in the experiment running all 19 problems concurrently. As the tasks include both regression and classification, the overall trend graph is the portion of correct outputs for given instances: predictions for classification and integer values for regression tasks.

We also separated the learning performances of agents and concatenated them in the order of increasing indices, following Table II, because the learning process of an agent only requires knowledge from agents with smaller indices. The next graph starts at the average number of instances that all agents on the left side (smaller indices) need to complete solving their problems. This format of illustration is also used in the next sections for comparisons between ConCS and other approaches.

The performance statistics of ConCS and XCSCF* are summarized in Table IV. When accumulating all agents, ConCS needs an average of 874 317 instances (standard deviation 284 341) in all agents to finish learning all problems. The longest run took up to 2 462 000 instances to complete solving all problems, while the fastest run took only 254 240 instances. For Hierarchical Carry-one and its subproblems, ConCS needs an average of 227 300 instances to find its optimal solution, which was longer than the 120 100 instances that XCSCF*

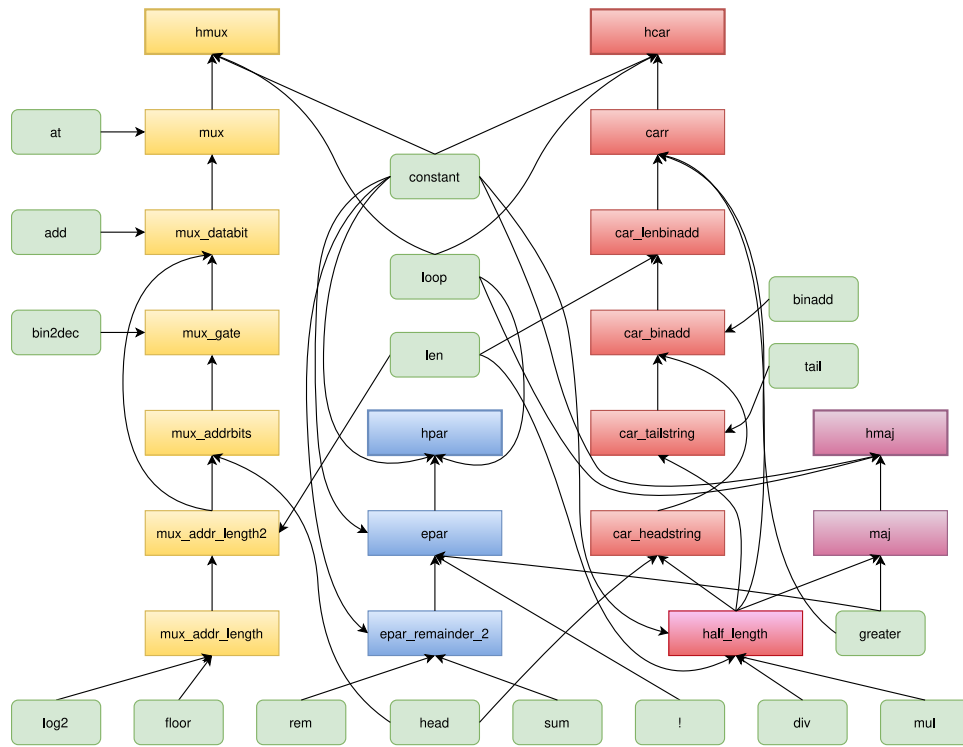


Fig. 3. Discovered network of knowledge by ConCS. See Tables II and III for the acronyms of the preprovided functions in green boxes and the learned functions (all boxes except green ones), respectively. These provide curricula to learn the target (top nongreen boxes) functions given initial axiomatic skills (green boxes).

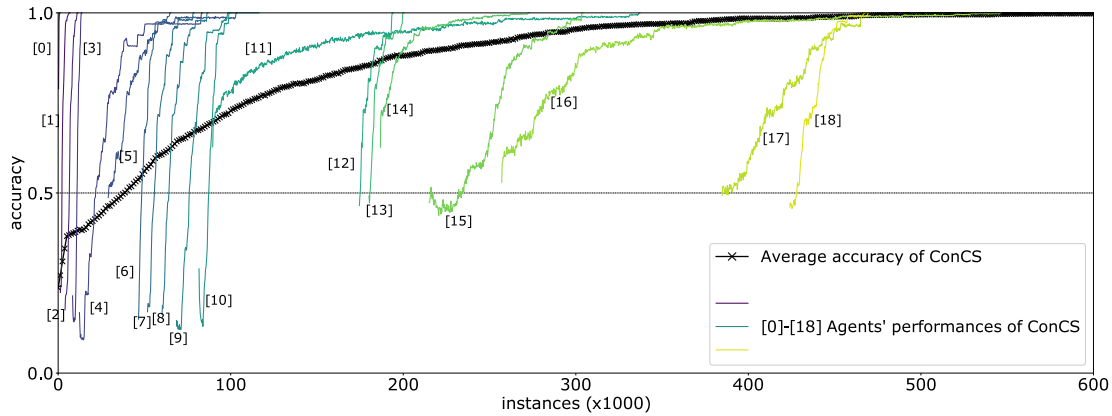


Fig. 4. Learning performances of ConCS. The Average graph depicts the portion of completely correct instances (accurate integer regression or classification) of 19 agents regarding the total experience of ConCS. Agent performances are the average learning curves of agents across runs which are ordered with increasing indices (see Table II) from agent indexed from [0] to [18]. The agent performances are plotted with their individual experience (instances). Starting points of these curves concatenate with the average number of completing instances of all their previous agents (on the left).

needed. XCSCF* learns faster than ConCS in all six curricula. This is expected with the curriculum learning overhead.

1) *Comparison With XCSCF*, XCS, and XCSCFC*: In this section, we will compare ConCS with XCSCF* using type-fitting XCSCFA [26], XCS, and XCSCFC. Because these three approaches were designed to learn a target problem, comparisons with them must be tested on specific problems instead of 19 problems with unrelatedness. The tests on XCSCF* and XCSCFC also follow their designed learning paradigms. For example, to test on Mux problems, while XCS learns a Mux problem at a specific scale, XCSCFC reuses CFs from Mux problems at lower scales. ConCS and XCSCF* both need a

set of components from subproblems before achieving the solutions for Mux problems. In terms of computation cost, ConCS and XCSCF* need fewer learning instances to solve the target problems compared with XCS [32] and XCSCFC [9] solving the 135-bit Mux problem even when including the numbers of instances for subproblems. Nevertheless, the population sizes of each individual system in ConCS and XCSCF* are much smaller than those of XCS and XCSCFC. Therefore, both ConCS and XCSCF* can finish learning the n -bit target problems swiftly for each run.

The first five problems from Table II are selected to test ConCS and XCSCF* as these are components of the

TABLE IV

PERFORMANCE STATISTICS OF CONCS ON GIVEN PROBLEMS. ALL NUMBERS ARE THE NUMBERS OF EXPLORED INSTANCES TO REACH OPTIMAL SOLUTIONS WITH 100% ACCURACY. H. IS HIERARCHICAL

Problems	ConCS		XCSCF*	
	Ave.	Std.	Ave.	Std.
All 19 problems	874,317	284,341	n/a	n/a
Multiplexer	71,400	29,468	39,800	24,915
Carry-one	80,667	24,980	52,817	20,244
H.Multiplexer	150,550	54,527	86,667	50,019
H.Carry-one	227,300	83,310	120,100	57,293
H.Majority-on	104,000	47,547	60,250	26,030
H.Even-parity	31,400	17,949	17,883	8146

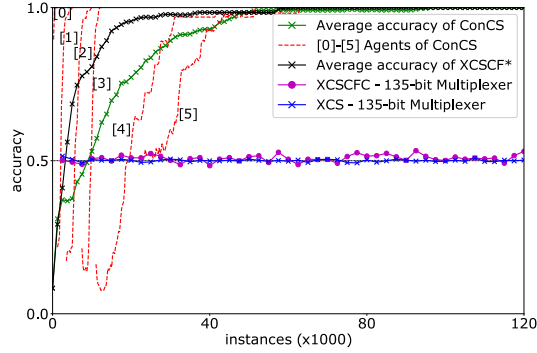


Fig. 5. Learning performances on the variable-scale Mux problem. Red curves depict the learning performances of agents in ConCS. These curves are arranged in the same manner with the agent performances in Fig. 4.

Mux domain. Fig. 5 shows four learning curves of the four approaches and the learning curves of ConCS agents in the same manner that Fig. 4 separates the component learning performances.

XCS and XCSCFC are tested on the Mux at a specific scale of 135 bits. In this experiment, XCSCFC learns 135-bit Mux problem with the identical configuration of Transfer Learning from 6-, 11-, 20-, 37-, and 70-bit Mux problems in [9]. We only compared with XCSCFC in this experiment because it can solve this problem at large scales (70 and 135 bits) while it cannot completely solve problems at relatively large scales of later experiments. We also did not compare with XCS in the Hierarchical Even-parity problem because XCS cannot scale well to this problem.⁵

According to Fig. 5, XCSCF* solves the variable-size Mux problem at a faster learning rate than ConCS, although both systems ultimately achieve the same classification/regression performance. This was because XCSCF* followed an LL style where the curriculum was fixed *a priori* while ConCS must determine the curriculum itself. If it is possible to provide correct human knowledge to the learning system XCSCF* will perform faster than without, but at the cost of having to have human knowledge in the first place. ConCS can generate the same solution as XCSCF* within 150 000 instances. ConCS and XCSCF* both outperform XCS and XCSCFC. This pattern of performance differences among the tested systems are analogous to those in other experiments for the sets of general

⁵Hierarchical Even-parity problem requires a one-to-one mapping of instances to rules in the solution of standard XCS.

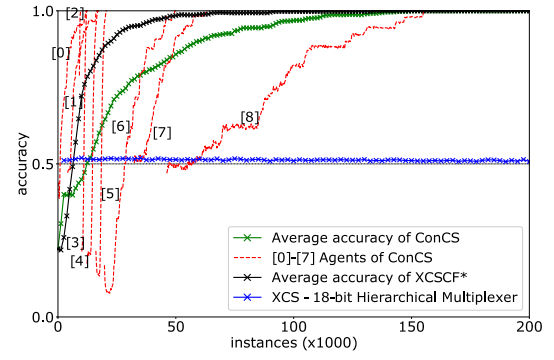


Fig. 6. Learning performances on the variable-scale Hierarchical Mux problem.

Hierarchical Mux (Fig. 6), Carry-one, Hierarchical Carry-one, Hierarchical Even-parity, and Hierarchical Majority-on problems (see the Supplementary material).

2) *Comparison With Other Machine Learning Algorithms:* In this section, we compare ConCS and XCSCF* with other graph-based machine algorithms, i.e., XGBoost (XGB) and random forest (RF), and standard GP on classifying large-scale and complex problems, such as Carry-one, Hierarchical Mux, Hierarchical Carry-one, Hierarchical Majority-on, and Hierarchical Even-parity problems. It is noted that, in these experiments, other methods were tasked with solving the problems directly without the provision of subproblems. The aim of the comparisons is to highlight ConCS performance when it is provided with subproblems.

Standard GP, XGB, and RF are normally used in supervised learning with separate training and testing sets. However, because ConCS can access any instance of the tested problems as its agents are online learning systems, we also experimented with the other methods having access to all possible instances. Because both ConCS and XCSCF* can solve the tested problems within less than 200 000 instances (one instance per iteration), we provided other methods with the same experience of 200 000 instances.⁶ Grid search was used to tune hyper-parameters of XGB and RF (not standard GP). The results are from the best parameters for each problem.

Results on Table V show that ConCS and XCSCF* [26] both achieve 100% accuracy in all problems. Because of the ability to solve tested problems at any scale, the accuracies of ConCS and XCSCF* are constantly 100%, which are significantly higher than the average accuracies of other methods in most problems (statistical significance based on the Wilcoxon signed rank test with *p-value* < 0.05). The differences will be increased if we enlarge the scales of the benchmark problems.

C. Continual Learning With Randomly Arriving Problems

This experiment shows the capability of ConCS to learn continually when the tasks arrive at different points in time. ConCS consistently solved all 19 problems in all 30 runs. ConCS was able to solve hard problems once the easier problems providing the necessary building blocks were presented

⁶For any 18-bit or higher scale Boolean problem, this is smaller than the total possible number of instances so overfitting is tested.

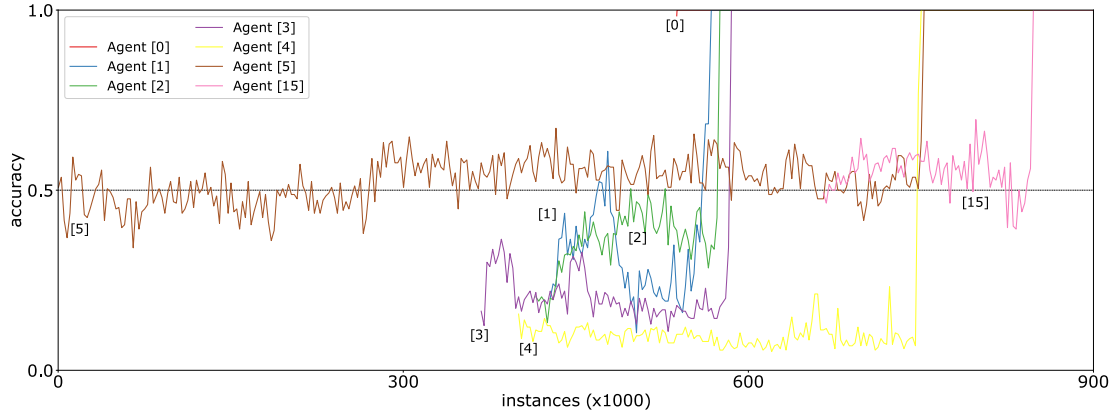


Fig. 7. Learning performance of agents related to the Hierarchical Mux domain when the problems are presented in a random order. Agents of ConCS were labeled using the “Id” column in Table II. The bigger the number, the more complex we consider the problem. Only problems [0-5] and [15], which are in the curriculum of the Hierarchical Mux domain, are shown for clarity.

TABLE V

ACCURACY COMPARISONS WITH OTHER MACHINE LEARNING APPROACHES: STANDARD (NAIVE) GP, XGB, AND RF. CONCS AND XCSCF*’S RESULTS ARE BOLD IF THEY ARE SIGNIFICANTLY HIGHER THAN THE RESULTS OF ALL OTHER BENCHMARK METHODS

Problems	ConCS	XCSCF*	GP	XGB	RF
16-bit Carry-one	1.0	1.0	0.959	0.998	0.994
18-bit H. Multiplexer	1.0	1.0	0.805	0.855	0.868
18-bit H. Carry-one	1.0	1.0	0.78	1.0	1.0
15-bit H. Majority-on	1.0	1.0	0.728	0.999	0.992
21-bit H. Majority-on	1.0	1.0	0.687	0.985	0.923
18-bit H. Even-parity	1.0	1.0	0.531	0.854	0.859

and solved. For clarity, Fig. 7 depicts only the learning curves of all agents related to the Hierarchical Mux problems. Problem 0, which was to find the address length given the Mux problem size (Section III-D), was presented later than most of the other problems. ConCS was gradually able to solve these other problems once the problem 0 was solved as it provides necessary building blocks to enable learning hard problems. This figure shows clearly that ConCS can learn continually by accumulating progressively more complex knowledge.

D. Experiments on Monk’s Problems

ConCS was also experimented on Monk’s problems 1 and 2 to validate its potential on real-valued problems. For Monk 1, we created two additional component subproblems to address two parts: $A_1 = A_2$ and $A_5 = 1$. For the experiment on Monk 2, a subproblem was added to address the following output $\sum(\text{equal}(\text{attlst}, 1))$. The system converged accurately on all runs with slight diversity on solutions for both Monk 1 and Monk 2. Figs. 8 and 9 show sample solutions of the two problems. The solution diversity is minor enough to easily infer the actual logic behind these problems, e.g., $\text{greater}(2, A_5)$ was discovered instead of $\text{equal}(A_5, 1)$.

V. DISCUSSION

ConCS can simultaneously solve a large number (19) of problem types, comprising a mix of regression and classification problems. ConCS can automatically determine the

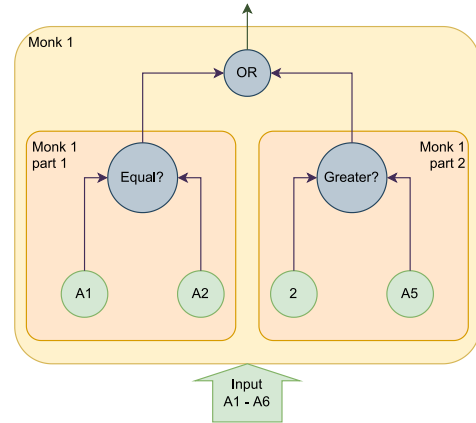


Fig. 8. ConCS’ sample solution on Monk 1 problem.

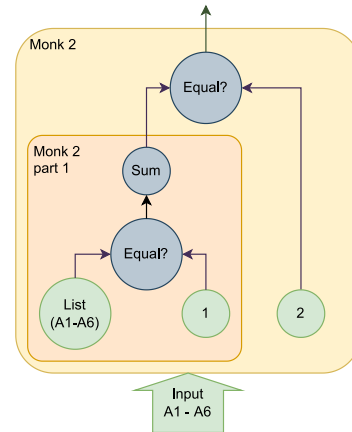


Fig. 9. ConCS’ sample solution on Monk 2 problem.

learning curricula, which need to be provided externally in XCSCF*. Replacing the human-in-the-loop from specifying the exact problems in exact order for continual learning with only needing to identify problems that might be relevant to each other without order is considered important to continual learning systems. Instead of fixed curricula to a known goal, open-ended problems without known curricula can be explored.

ConCS was slower to reach optimum performance in benchmark problems compared with XCSCF*. Efficiency is lost to gain the ability to continually learn without the need for human bootstrapping of the curricula. This result is predictable because each learning process can only commence when all prerequisite building blocks are available, where ConCS has to determine itself which agents have all the necessary building blocks available unlike other systems that are provided with this order. In separate experiments, XCS and XCSCFC (in the Mux domain only) can only solve tested problems at limited scales. In contrast, both ConCS and XCSCF* produce general solutions, which can solve tested problems at infinite scales (any scale within the limit of computation hardware) with 100% accuracy. Therefore, if we choose a problem at a large enough scale as the benchmark, ConCS, and XCSCF* will always outperform XCS and XCSCFC because prediction performances of ConCS and XCSCF* are not affected by the problem scale.

A major strand of future work will be the introduction of real-valued domains into ConCS. As CFs are tree-based programs they can accept real values as features into their leaf nodes. Thus, CFs have been adapted to image classification [57] and robot navigation [58] with ongoing work in linguistics due to their semantic compositionality (capacity to combine structures into sequences with learned meanings, such that the meaning is a function of the meanings of its parts [59]). The varied features in the root nodes are passed in the CFs to appropriately typed functions that ultimately return a Boolean yes/no at the root node of the CF tree.

CF functions and their hyper parameters in the nodes of trees can be deterministic, stochastic, or probabilistic. They can be case-based (as in decision trees), loops (as in recursive functions), or state machines, where how to set default values, stop infinite loops, and ensure terminating state machines, respectively, are open questions. Furthermore, CFs enable symbol productivity, i.e., unbounded expressive power through finite means, compacting the representation of knowledge [59].

Arguably the biggest issue faced for the adoption of real-valued problems is the creation of early curriculum training problems/environments that can help guide the learning system. Thus, an issue is a lack of appropriate datasets to continually build functions plus the related CFs with encapsulated useful knowledge—an analogy is the need for children’s books rather than teaching directly from the Encyclopædia Britannica. The focus shifts from designing algorithms to designing training environments with interesting problems that lead to reusable knowledge/functionality. Co-evolution of environments and agents is possible, cf POET [60], although this may be less straightforward than creating multiple environments with preseeded properties that could become useful to the agent as its problem-solving abilities improve.

Next is ConCS generalizes well (removes redundant/irrelevant information) but only partially abstracts information (identifies higher order patterns from previous problems), where the introduction of a lateralized architecture will enable the system to consider the current input at both local and holistic levels simultaneously [58]. Currently, a

rule-base needs to achieve perfect performance prior to being reused as a function, which would not be possible in noisy or continuous data domains. In future, not only would it be possible to (re)use emerging rule-bases as functions, but it is plausibly beneficial as the worth of each alternative rule-base could be measured both on the performance in its domain and its ability to improve solutions in other problems where its functionality is encapsulated in their CF nodes.

ConCS, XCSCF*, and GP with tree programs are capable of encoding precisely the patterns of target tasks, while XGB and RF aim to approximate the output. The advantage of ConCS and XCSCF* is the ability of GP-like trees to encode complex knowledge with the use of provided functions and ruleset functions. GP-like trees are more flexible than XGB and RF’s graphs that rely on linear separation. With constituent patterns obtained from continual learning, ConCS can combine the tree-based piecewise knowledge to construct novel decision boundaries in any nonlinear/linear combinations that accumulated knowledge enables.

Arguably, ConCS and XCSCF* solved tested problems more effectively than standard GP because these two systems were provided with the subtasks representing problem components. However, the provision of the subproblems is analogous to the way young children build up their intelligence with sets of lessons that have been optimized over human civilization. It is possible that GP with proper modifications and subproblems can also capture the complex logic of tested problems.

Experimental results on Boolean problems showcase the capability of using tree-based LCSs to learn continually and accumulate complex knowledge. Other categories of tasks might require different approaches of adapting continual or layered LCSs (ConCS/XCSCF*), such as using another encoding method, such as neural networks or another complexity-growing approach, such as the online feature-generation module in XOF [46], [47].

It is interesting to consider the computational complexity of a system that seeks to produce a general solution to a problem. Wilson, and then Butz, conclude that XCS scales polynomially in time and space complexity [61], [62], where all systems considered here inherit their core functionality from this framework. The additional type-fitting and communication between interacting populations are again considered polynomial. However, once ConCS has discovered a general solution to a problem, this will scale depending on the patterns within the solution, rather than the number of features within the input.

VI. CONCLUSION

In this study, we developed ConCS as the first system that can solve multiple Boolean problems continually without human-developed curricula. The minimal human involvement is to provide axiomatic knowledge and useful subproblems, where more than necessary can be provided—the system learns but does not reuse unrelated problems. This enabled solving complex Boolean problems at scales (any n -bit scale) beyond conventional approaches.

The novelty that enables ConCS to learn continually is a system of LCSs utilizing type-fitting tree-based programs to encode high-level knowledge in a pool to store/reuse accumulated knowledge. Type-fitting trees enable ConCS to capture complex knowledge behind target problems, which was shown to have the potential to extend beyond Boolean domains. Moreover, through learning continually with parallel tasks and subtasks, ConCS can construct novel knowledge by combining flexibly preprovided functions and constituent patterns learned in subtasks.

Thanks to the apparent representation of the learned tree-based rules, it is straightforward to formulate a network of knowledge among problems. The network connections enable effective references to only relevant knowledge. This characteristic is important in an AI system with a huge volume of accumulated knowledge, where checking all knowledge is impractical. Moreover, the resulted knowledge network can yield learning curricula, which was previously guided by humans in LL [17], [26]. This new capacity increases the autonomy of AI systems as human guidance may not be always available.

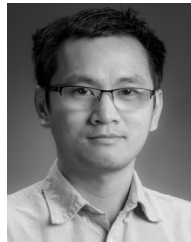
In certain cases, learned knowledge provided by ConCS delivers an unexpected understanding of target problems, which can be surprisingly simple. Even with an increasing volume of knowledge that leads to increasing search spaces, the problem-solving capability of ConCS keeps being built up by acquiring progressively more complex functions.

The current implementation of ConCS relies on a single physical computation unit where prioritization of agents is necessary. Utilizing distributed LCS frameworks [63] and hardware could accelerate the learning process by distributing the computation into multiple physical computing units. Parallelizing the computation also enables accumulating a larger volume of knowledge to assist in solving a greater range of problems.

REFERENCES

- [1] J. Piaget and M. Cook, *The Origins of Intelligence in Children*. New York, NY, USA: Int. Univ. Press, 1952.
- [2] S. Thrun and T. M. Mitchell, "Learning one more thing," Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, USA, Rep. TR-1994, Sep. 1994.
- [3] S. Thrun and L. Pratt, *Learning to Learn*. New York, NY, USA: Springer, 2012.
- [4] P. H. Stone, *Layered Learning in Multiagent Systems*. Carnegie Mellon Univ., Pittsburgh, PA, USA, 1998.
- [5] M. B. Ring, *Child: A First Step Towards Continual Learning*. Boston, MA, USA: Springer, 1998, pp. 261–292.
- [6] S. Whiteson and P. Stone, "Concurrent layered learning," in *Proc. 2nd Int. Joint Conf. Auton. Agents Multiagent Syst. (AAMAS)*, 2003, pp. 193–200.
- [7] D. Potts and B. Hengst, "Concurrent discovery of task hierarchies," in *Proc. AAAI Spring Symp. Knowl. Rep. Ontol. Auton. Syst.*, 2004, pp. 1–8.
- [8] B. Liu, "Lifelong machine learning: A paradigm for continuous learning," *Front. Comput. Sci.*, vol. 11, no. 3, pp. 359–361, 2017.
- [9] M. Iqbal, W. N. Browne, and M. Zhang, "Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems," *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 465–480, Aug. 2014.
- [10] Y.-S. Ong and A. Gupta, "Evolutionary multitasking: A computer science view of cognitive multitasking," *Cogn. Comput.*, vol. 8, no. 2, pp. 125–142, 2016.
- [11] A. Gupta, Y.-S. Ong, and L. Feng, "Insights on transfer optimization: Because experience is the best teacher," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 1, pp. 51–64, Feb. 2018.
- [12] L. Bull, "A brief history of learning classifier systems: From CS-1 to XCS and its variants," *Evol. Intell.*, vol. 8, nos. 2–3, pp. 55–70, 2015.
- [13] S. W. Wilson, "Classifier fitness based on accuracy," *Evol. Comput.*, vol. 3, no. 2, pp. 149–175, Jun. 1995.
- [14] R. J. Urbanowicz and J. H. Moore, "Learning classifier systems: A complete introduction, review, and Roadmap," *J. Artif. Evol. Appl.*, vol. 2009, no. 1, pp. 1–25, 2009.
- [15] R. Urbanowicz, A. Granizo-Mackenzie, and J. Moore, "Instance-linked attribute tracking and feedback for michigan-style supervised learning classifier systems," in *Proc. 14th Annu. Conf. Genet. Evol. Comput. (GECCO)*, 2012, pp. 927–934.
- [16] I. M. Alvarez, W. N. Browne, and M. Zhang, "Reusing learned functionality in XCS: Code fragments with constructed functionality and constructed features," in *Proc. Companion Annu. Conf. Genet. Evol. Comput. (GECCO)*, 2014, pp. 969–976.
- [17] I. M. Alvarez, W. N. Browne, and M. Zhang, "Human-inspired scaling in learning classifier systems: Case study on the n -bit multiplexer problem set," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, 2016, pp. 429–436.
- [18] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proc. 26th Annu. Int. Conf. Mach. Learn. (ICML)*, 2009, pp. 41–48.
- [19] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," in *Proc. Int. Conf. Learn. Rep. (ICLR)*, 2014, pp. 1–9.
- [20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [21] N. Dikkala, G. Kaplun, and R. Panigrahy, "For manifold learning, deep neural networks can be locality sensitive hash functions," 2021, *arXiv:2103.06875*.
- [22] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, p. 19.
- [23] J. Holland, *Adaptation in Artificial and Natural Systems*. Ann Arbor, MI, USA: Univ. Michigan Press, 1975.
- [24] R. J. Urbanowicz and W. N. Browne, *Introduction to Learning Classifier Systems*, Springer, 2017.
- [25] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends Cogn. Sci.*, vol. 3, no. 4, pp. 128–135, 1999.
- [26] I. M. Alvarez, T. B. Nguyen, W. N. Browne, and M. Zhang, "A layered learning approach to scaling in learning classifier systems for Boolean problems," 2020, *arxiv:abs/2006.01415*.
- [27] R. Confalonieri, L. Coba, B. Wagner, and T. R. Besold, "A historical perspective of explainable artificial intelligence," *Interdiscipl. Rev. Data Min. Knowl. Disc.*, vol. 11, no. 1, 2021, Art. no. e1391.
- [28] M. V. Butz, *Rule-Based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design*. Berlin, Germany: Springer-Verlag, 2007.
- [29] A. Tomlinson and L. Bull, "A corporate classifier system," in *Proc. Int. Conf. Parallel Problem Solving Nat.*, 1998, pp. 550–559.
- [30] A. Tomlinson and L. Bull, "An accuracy based corporate classifier system," *Soft Comput.*, vol. 6, nos. 3–4, pp. 200–215, 2002.
- [31] M. V. Butz and S. W. Wilson, "An algorithmic description of XCS," in *Advanced in Learning Classifier Systems (LNAI 1996)*, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds. Berlin, Germany: Springer-Verlag, 2001, pp. 253–272.
- [32] M. Nakata and W. N. Browne, "Learning optimality theory for accuracy-based learning classifier systems," *IEEE Trans. Evol. Comput.*, vol. 25, no. 1, pp. 61–74, Feb. 2021.
- [33] A. Giani, F. Baiardi, and A. Starita, "PANIC: A parallel evolutionary rule based system," in *Proc. Evol. Program.*, 1995, pp. 753–771.
- [34] L. Bull and T. O'Hara, "Accuracy-based neuro and neuro-fuzzy classifier systems," in *Proc. 4th Annu. Conf. Genet. Evol. Comput. (GECCO)*, 2002, pp. 905–911.
- [35] P. L. Lanzi and D. Loiacono, "XCSF with neural prediction," in *Proc. IEEE Int. Conf. Evol. Comput.*, 2006, pp. 2270–2276.
- [36] H. H. Dam, H. A. Abbass, C. Lokan, and X. Yao, "Neural-based learning classifier systems," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 1, pp. 26–39, Jan. 2008.

- [37] G. Howard, L. Bull, and P.-L. Lanzi, "A spiking neural representation for XCSF," in *Proc. IEEE Congr. Evol. Comput.*, Jul. 2010, pp. 1–8.
- [38] K. Matsumoto et al., "Learning classifier system with deep autoencoder," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2016, pp. 4739–4746.
- [39] M. Ahluwalia and L. Bull, "A genetic programming-based classifier system," in *Proc. Genet. Evol. Comput. Conf.*, vol. 1, 1999, pp. 11–18.
- [40] P. L. Lanzi and P. Alessandro, "Extending the representation of classifier conditions—Part II: From messy coding to S-expressions," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, 1999, pp. 345–352.
- [41] P. L. Lanzi, "XCS with stack-based genetic programming," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, vol. 2, 2003, pp. 1186–1191.
- [42] C. Ioannides and W. Browne, "Investigating scaling of an abstracted LCS Utilising ternary and S-expression alphabets," in *Proc. Genet. Evol. Comput. Conf. (Companion)*, 2007, pp. 2759–2764.
- [43] M. Iqbal, W. N. Browne, and M. Zhang, "Evolving optimum populations with XCS classifier systems," *Soft Comput.*, vol. 17, no. 3, pp. 503–518, 2013.
- [44] M. Iqbal, W. N. Browne, and M. Zhang, "Extending learning classifier system with cyclic graphs for scalability on complex, large-scale boolean problems," in *Proc. 15th Annu. Conf. Genet. Evol. Comput. (GECCO)*, 2013, pp. 1045–1052.
- [45] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA, USA: MIT Press, 1994.
- [46] T. B. Nguyen, W. N. Browne, and M. Zhang, "Online feature-generation of code fragments for XCS to guide feature construction," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, 2019, pp. 3308–3315.
- [47] T. B. Nguyen, W. N. Browne, and M. Zhang, "Improvement of code fragment fitness to guide feature construction in XCS," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, 2019, pp. 428–436.
- [48] T. B. Nguyen, W. N. Browne, and M. Zhang, "Relatedness measures to aid the transfer of building blocks among multiple tasks," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, 2020, pp. 1–7.
- [49] P. Stone and M. Veloso, "Layered learning," in *Proc. Mach. Learn. (ECML)*, 2000, pp. 369–381.
- [50] A. Siddique, W. N. Browne, and G. M. Grimshaw, "Frames-of-reference based learning: Overcoming perceptual aliasing in multi-step decision making tasks," *IEEE Trans. Evol. Comput.*, vol. 26, no. 1, pp. 174–187, Feb. 2022.
- [51] B. Da, A. Gupta, and Y.-S. Ong, "Curbing negative influences online for seamless transfer evolutionary optimization," *IEEE Trans. Cybern.*, vol. 49, no. 12, pp. 4365–4378, Dec. 2019.
- [52] P. Tommasino, D. Caligiore, M. Mirolli, and G. Baldassarre, "A reinforcement learning architecture that transfers knowledge between skills when solving multiple tasks," *IEEE Trans. Cogn. Develop. Syst.*, vol. 11, no. 2, pp. 292–317, Jun. 2019.
- [53] G. Sun, C. Yang, J. Liu, L. Liu, X. Xu, and H. Yu, "Lifelong metric learning," *IEEE Trans. Cybern.*, vol. 49, no. 8, pp. 3168–3179, Aug. 2019.
- [54] E. Real, C. Liang, D. So, and Q. Le, "AutoML-Zero: Evolving machine learning algorithms from scratch," in *Proc. 37th Int. Conf. Mach. Learn.*, vol. 119, Jul. 2020, pp. 8007–8019.
- [55] R. Caruana, "Multitask learning," *Mach. Learn.*, vol. 28, no. 1, pp. 41–75, 1997.
- [56] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.
- [57] A. Siddique, W. N. Browne, and G. M. Grimshaw, "Lateralized learning for robustness against adversarial attacks in a visual classification system," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, 2020, pp. 395–403. [Online]. Available: <https://doi.org/10.1145/3377930.3390164>
- [58] A. Siddique, W. N. Browne, and G. M. Grimshaw, "Frames-of-reference-based learning: Overcoming perceptual aliasing in multistep decision-making tasks," *IEEE Trans. Evol. Comput.*, vol. 26, no. 1, pp. 174–187, Feb. 2022.
- [59] J. A. Fodor and Z. W. Pylyshyn, "Connectionism and cognitive architecture: A critical analysis," *Cognition*, vol. 28, no. 1-2, pp. 3–71, 1988.
- [60] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, "POET: Open-ended coevolution of environments and their optimized solutions," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, 2019, pp. 142–151.
- [61] S. W. Wilson, "Generalization in the XCS classifier system," in *Proc. 3rd Annu. Conf. Genetic Program*, J. R. Koza, et al., Eds. 1998, pp. 665–674.
- [62] M. V. Butz, D. E. Goldberg, and P. L. Lanzi, "Computational complexity of the XCS classifier system," in *Foundations of Learning Classifier Systems* (Studies in Fuzziness and Soft Computing), L. Bull and T. Kovacs, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 91–126.
- [63] H. H. Dam, P. Rojanavasu, H. A. Abbass, and C. Lokan, "Distributed learning classifier systems," in *Learning Classifier Systems in Data Mining*. Heidelberg, Germany: Springer-Verlag, 2008, pp. 69–91.



Trung B. Nguyen (Member, IEEE) received the Ph.D. degree in computer science from the Victoria University of Wellington, Wellington, New Zealand, in 2021.

His Ph.D. research was on the collaboration of multiple learning classifier systems using code fragments to learn complex Boolean problems as well as real-world datasets with more autonomy than existing systems. He has published a number of academic papers in high-ranked international conferences and journals.

Dr. Nguyen is a member of the IEEE Computational Intelligence Society.



Will N. Browne (Member, IEEE) received the Doctoral degree in industrial learning classifier systems (LCSs) from the University of Wales, Cardiff, U.K., in 1999.

His Postdoctoral work was with the University of Leicester, Leicester, U.K., followed by eight years lecturing in Cybernetics, University of Reading, Reading, U.K. He led the LCS theme with the Evolutionary Computation Research Group, Victoria University of Wellington, Wellington, New Zealand, from 2009 to 2021. He is currently a Professor and the Chair of Manufacturing Robotics from the Queensland University of Technology, Brisbane, QLD, USA. He has coauthored the first textbook on LCSs *Introduction to Learning Classifier Systems* (Springer, 2017). His research focuses on applied cognitive systems, including cognitive robotics, learning classifier systems, and explainable artificial intelligence.

Prof. Browne has been the Co-Track Chair for the evolutionary machine learning tracks and provided tutorials on rule-based machine learning at major conferences.



Mengjie Zhang (Fellow, IEEE) received the B.E. and M.E. degrees from the Artificial Intelligence Research Center, Agricultural University of Hebei, Baoding, China, in 1989 and 1992, respectively, and the Ph.D. degree in computer science from RMIT University, Melbourne, VIC, Australia, in 2000.

He is currently a Professor of Computer Science with the Victoria University of Wellington, Wellington, New Zealand. He has published over 800 research papers.

Prof. Zhang is a Fellow of the Royal Society and Engineering of New Zealand, and an IEEE Distinguished Lecturer.