Resilient Supervisory Multi-Agent Systems

Kleio Baxevani, *Student Member, IEEE*, Ashkan Zehfroosh, *Student Member, IEEE*, and Herbert G. Tanner, *Senior Member, IEEE*

***Abstract*—Accidental or deliberate disruption of the coordination function in a multi-agent system has been discussed and referred to in the social sciences literature as *leader decapitation*; this paper outlines a methodology for making multi-agent networks resilient to this type of failure, enabling a timely restoration of operation normalcy by leveraging machine learning techniques. The approach involves endowing the agents with a cascade of independent learning modules that enable them to discover over time their role in the overall system coordinating strategy, so that they are able to autonomously implement it when central coordination seizes to function. Through these machine learning algorithms, the agents incrementally identify the overall system's task specification and simultaneously optimize their strategy to serve the common goal.**

I. Introduction

The problem of designing resilient multi-agent systems is pervasive and can be identified in application instances in manufacturing, building energy management, the smart grid, and self-driving cars [1], [2], among several others. The need to develop novel design and control paradigms that go beyond the traditional notions of robustness, reliability, and stability, has also been recognized [3].

Figure 1 shows an instance of a robotic-assisted pediatric rehabilitation study [4] where a team of (heterogeneous) robots interact socially with an infant who has motor delay, in an effort to encourage and entice physical activity which is a catalyst for both motor and cognitive development at this age. The robots' social interaction with the human subject is coordinated through an optimal strategy produced by a reinforcement learning algorithm, which takes as inputs prior data for instances of infant reactions to robot actions, and optimizes for infant engagement and motor response in response to robot actions.

Here, the robots are centrally coordinated, as none can have a holistic view of what is happening in the scene in order to optimize its behavior.

The application study of Fig. 1 is an instance of a general case where a multi-robot system is coordinated and synchronized via a centralized decision-maker. This coordinator can also be described as the event-based dynamics of the swarm that are responsible for managing/coordinating the swarm, and the agents belong to the time-based dynamics responsible for the execution of the plan as presented in [5]. A well-recognized limitation of such a control architecture is the existence of a single point of failure: if the central decision-maker (the coordinator) is somehow taken off-line, the system is paralyzed.

The authors are with the Department of Mechanical Engineering, University of Delaware. Email: {kleiobax,ashkanz,btanner}@udel.edu

This work had been supported in part by NIH under grant # R01HD87133 and by NSF's SCH program via award # 2014264.

Fig. 1. Snapshot of an infant within the GEAR system socially interacting with multiple robots.

Although resilience can emerge as a result of judicious design of interacting agent objectives and incentives, it is argued [3] that much like any mission-driven organization will fail without organizational leadership, a supervisory design is still needed to ensure smooth operation. But when the leader in such a multi-agent system seizes to function, how can one prevent the whole system from collapsing *and* recover normalcy of operation? To answer this question, the paper considers a class of supervisory systems, and proposes a novel resilient design paradigm that incorporates learning modules to enable the system to recover normalcy of operation following a leader (coordinator) "decapitation." It is inspired by how New York City's emergency management center was recreated on site by means of grass-root, spontaneous efforts of citizens, after it had collapsed with the twin towers [6]. The approach integrates machine learning modules based on reinforcement learning combined with formal languages and grammatical inference [7], within the subordinate control units (the individual robots).

This paper argues that machine learning be utilized to make such architectures resilient to coordinator decapitation. What is more, it extends recent results on this problem, and further promotes resilience by distributing decision-making functions to the individual agents and enable them to locally optimize their behaviors in concert with the mechanism they use to become resilient to coordinator failure.

Technically, the idea promoted here is for the robotic agents to keep track of the actions they perform in support of the overall coordination strategy (which is unknown to them) during normal operation. This is key to being able to understand what needs to be done in order to employ a recovery contingency algorithm which will enable them to infer the strategy and essentially collaboratively "clone" their supervisory decision-making module. In this way, the agents cooperate to ensure the four key elements safeguarding against destabilization after leader decapitation [8]: (i) information ow, (ii) consensus-reaching ability, (iii) functional capability, and (iv) information interpretation. This approach deviates from the one reported in earlier work [9] along this direction in several key aspects: rst, the agents are far more autonomous as they formulate (under guidance) their own action plan using reinforcement learning; second, the agent action plans are now optimal relative to some performance metric; third, communication with the system's coordinator is bidirectional; and nally, security is enhanced because the (optimal) group action plan is not directly communicated from the coordinator to the agents. In addition, the reinforcement learning (R) methodology implemented on the agents in this paper is distinct from the realization in the algorithm's original debut [10] because here the agents are neither aware of their reward function nor of their actual dynamics; this missing information is unveiled progressively in the form of agent supervision by the overall system's coordinating entity. What is perhaps more interesting is that for the class of target task speci cation languages considered in the implementation study of this paper, the R algorithm catalyzes a dramatically more ef cient solution to an otherwise dif cult problem of language identi cation.

This paper contributes to the area of resilient multi-robot systems by uniquely combining distributed reinforcement learning with grammatical inference to achieve resilience to targeted attacks. In the approach reported here, (a) agents autonomously learn optimal policies for achieving their collective objective under the distant guidance of a supervising authority (Section V-C); (b) use the byproduct of their distributed optimal control synthesis algorithms to power a different learning module which practically performs system identi cation on their system task speci cation (Algorithm 1, line 17); and nally (c) combine their local task speci cation hypotheses to recreate their supervising process (Section V-D). Thus, as it will be shown in Section V-D, robotic agents that identify their own task speci cation language based on local information, can subsequently perform grammatical inference and then "compare notes" to infer the originally unknown global system speci cation language. In this way, should the coordinator goes of ine for any reason, the system will still be able to function normally, therefore realizing elements of resilience observed in human collectives [6].

The rest of the paper is organized as follows. Section II reviews elements of related literature within the domains of multi-agent system resilience, reinforcement learning, and grammatical inference. Some necessary mathematical background on the latter two topics is presented brie y in Section III and the problem description is presented in Section IV. Section V contains the main body of the analysis and results of this paper, while it also presents in detail a case study that (numerically) demonstrates the ef cacy of the reported methodology, and provides some additional insights. The paper closes with a quick overview in Section VI.

II. Related Literature

One perspective in the existing literature on how resilience manifests itself in multi-agent systems is related to the robustness of the communication network. In the presence of malicious robots, preserving a formation within a multi-robot system can be achieved through algorithms that construct communication graphs with the smallest number of nodes [11]. An essential element to this approach, is a method which guarantees that a topological network property (speci cally, r-robustness) is kept above a critical resilience threshold [12]. Due to high connectivity often required for such networks, and delays on receiving information, this can be a challenging task, and different protocols have been designed for synchronous and asynchronous time varying communication graphs [13], [14], [15], [16]. From the input-to-output stability standpoint, the resilience of consensus networks can be examined through a non-singular linear transformation that exposes the disturbance rejection performance of the system [17]. In the context of resilience for heterogeneous multi-agent systems, some work has been done studying the case of failure of single agents [18]. It is suggested that the network recon gures itself to maintain its original area coverage without increasing its connectivity. Another approach to render a heterogeneous multi-agent swarm more resilient to malicious agent actions is by monitoring the state values of the agents [19], and ignoring input from neighboring units if it is signi cantly different.

In swarms, decision-making is completely decentralized by design, so it comes as no surprise that only a small set of relatively simple emergent behaviors can be observed. In different engineering context, however, multi-agent collections may be required to exhibit much more sophisticated and adaptive behaviors—examples include applications related to privacy and security [20]. Such behaviors usually require some level of centralized decision-making—which motivates a coordination architecture similar to the one used in the application of Fig. 1. It is important that these systems are able to maintain normalcy and restore function following a failure (or a malicious attack in the context of national/cyber-security). Making a system resilient to such events is nontrivial due to the high degree of inter-connectivity among the physical and software components, and the intricate cyber, cognitive and human inter-dependencies [21].

Studies on centrally coordinated networked system resilience after leader decapitation have also been conducted within social and political sciences, focusing on counter-terrorism tactics [8] and cyber-security defense [22]. In general, proliferating the organization's coordinating plans and strategies throughout all the agents implementing the strategy (which would otherwise make sense from a robustness standpoint) creates multiple security vulnerabilities and is considered detrimental to security and operational integrity. An attacker would be able to exploit a vulnerability at any of the distributed agent sites to gain access and insight into how the whole organization is structured and controlled. In fact, even in the case of the motivating application of Fig. 1, distributing the planning capability among the agents does not necessarily improve resilience, because the actions of the agents may be *interdependent*. Then if both robots are involved in gameplay with the infant and one robot fails, then the plan may be at risk.

Alternatively, the strategy algorithm can be maintained in a single, remote, and secured physical device. Such distributed architectures that include physically separate and private communication channels between unsecured and trusted processes, are the hallmark of separation kernels used in cryptography and secure system design [23], and have become more increasingly more prevalent given the trend for miniaturization of communication devices. Moreover, it has been argued that they facilitate formal verification especially in cases of isolated channels with different security levels [24].

Along this direction one approach to making supervisory multi-agent systems more resilient to targeted attacks or centralized failures [9] suggests that in order to avoid system-level failure after leader decapitation, all agents need to learn how they to behave in the context of the (unknown to them) strategy their coordinator is trying to implement. In that approach, the strategy itself was given (or centrally devised) and the agents were passive executors. The paper at hand departs from this line of thought by decentralizing strategy formation to some extent, and by shifting some decision-making down to the level of the agents, which *learn* optimal policies for contributing toward the system-level task specification.

One proven approach to learn and form strategies under uncertainty is *reinforcement learning* (RL) [25]. While several RL algorithms may apply to a learning problem, few can guarantee convergence rates as a function of the amount of training data. Among the ones that do, are those which are classified as probably asymptotically correct (PAC). Existing PAC algorithms can be broadly divided into two groups: model-based algorithms like [26], [27], [28], [29], and model-free algorithms [30], [31]. Each group has its advantages and disadvantages. Model-based RL is usually more efficient when the state-space size of the system is not relatively large, while the efficiency of model-free RL is much more in systems with huge state-space size [32].

Neurophysiologically-inspired hypotheses [33] have suggested that the brain approaches complex learning tasks either in a model-free (trial and error), or model-based (deliberate planning and computation) fashion, or even combination of both, depending on the amount and reliability of the available information. This combination is postulated to contribute to making the process efficient and fast [34]. A fast learning process is particularly important in the motivating application of Section I (Fig. 1), since learning data on infant behavior as they interact socially with robots are sparse and can rarely be aggregated [35]. Taking the aforementioned considerations into account, this paper reports on the development of a new hybrid PAC algorithm called Dyna-Delayed Q-learning (DDQ) [10], which judiciously combines two PAC algorithms: model-based R-max and model-free Delayed Q-learning. It can be shown that DDQ not only inherits the best of the both worlds, but also outperforms its constituent technologies in most cases [10].

This paper is not the first where reinforcement learning is adopted to achieve resilience on a multi-agent system; off-policy reinforcement learning has been applied to learn the optimal solution to the synchronization problem in the presence of attacks and system uncertainties [36]. In this approach, similarly to ours, the knowledge of the agent's dynamics is not required. Another instance where learning is utilized to promote resilience in a multi-agent system is a work where the problem is formulated as a cooperative-competitive game [37], in which the protagonists represent the target agents, and the antagonists, the failures of the system. The approach in the present paper, however, is unique in the sense that learning is not merely a component of the solution; it *is* the solution.

III. TECHNICAL PRELIMINARIES

A brief description of learning techniques for formal languages, the systems, and the class of languages we consider in this work follows next. The introduced terminology is then used to describe technically the problem tackled here.

A. Formal languages

An *alphabet* is a finite set of symbols; here, alphabets are referred to with capital Greek letters (Σ or Λ). A *string* is a finite concatenation of symbols, σ, taken from an alphabet Σ. In this sense, strings are "words," formed as combinations of "letters," within a finite alphabet. A string u is of the form

$$u = \sigma_0\sigma_1\sigma_2\cdots\sigma_n \quad \text{such that each } \sigma_i \in \Sigma .$$

For a string w let $|w|$ denote its length. The *empty string* λ is the string of length 0. For two strings u, v, uv denotes their concatenation. Let Σ^* denote the set of all strings (including λ) over alphabet Σ, and Σ^n all strings of length n over Σ. For strings $v, w \in \Sigma^*$, v is a *substring* of w if there exist some $u_1, u_2 \in \Sigma^*$ such that $u_1 v u_2 = w$. The *k-factors* of a string w, denoted $f_k(w)$, are its substrings of length k. Formally,

$$f_k(w) = \begin{cases} \{u \in \Sigma^k \mid u \text{ is a substring of } w\}, & \text{if } |w| \geq k \\ \{w\}, & \text{otherwise} \end{cases} .$$

Subsets of Σ^* are called stringsets, or *languages*. By default, all languages considered here are assumed to contain λ. A *grammar* is a finite representation of a (potentially infinite) language. For a grammar G, let $L(G)$ denote the language generated by G. A *class* of languages $\mathcal{L}$ is a set of languages, e.g., the set of languages describable by a particular type of grammar.

In its application example, this paper will make use of the Locally k-Testable class of languages [38], [39]. A language L is Locally k-Testable if there is some k such that, for any two strings $w, v \in \Sigma^*$, if $f_k(w) = f_k(v)$ then either both w and v are in L or neither are. Thus a Locally k-Testable language is one for which membership in that language is decided entirely by substrings of length k.

For example, let $\Sigma = \{a, b\}$ and L_{bb} be the set of strings over Σ which contain at least one bb substring. In other words,

$$L_{bb} = \{bb, abb, bba, bbb, aabb, abba, abbb, \ldots\} .$$

Language L_{bb} is Locally 2-Testable because for any $w \in \Sigma^*$, whether or not w is a member of L_{bb} can be determined by seeing if $f_2(w)$ contains bb. In fact, L_{bb} belongs to a subclass of the Locally k-Testable languages for which any language in the subclass can be described by a grammar G which contains a single required k-factor; i.e., $L(G) = \{w \mid G \in f_k(w)\}$. In this case, L_{bb} is $L(G)$ for $G = \{bb\}$. This particular subclass is used here in the context of application examples, since its member languages can be learned from positive data in a straightforward way, as described below.

B. Language identification in the limit

The learning paradigm used in this work is that of *identification in the limit from positive data* [40]. The particular definition here is adapted from earlier work (cf. [41]): given a language L, a *presentation* of L is a function $: \mathbb{N} \to L \cup \{\#\}$, where $\#$ is a symbol not in and is used just to mark a location in the presentation with no data—these locations mark the beginning or end of words in the text. Then is a *positive presentation* of L if for all $w \in L$, there exists $n \in \mathbb{N}$ such that $(n) = w$.

Let $[i]$ denote the sequence $(0); (1); \ldots; (i)$. A learner or grammatical inference module ($\textsc{gim}$) is an algorithm which takes such a sequence as an input and outputs a grammar. A learner is said to *converge* on a presentation if there is some $n \in \mathbb{N}$ that for all $m > n$, $\textsc{gim}([n]) = \textsc{gim}([m])$.

A learning $\textsc{gim}$ is said to *identify a class* $\mathcal{L}$ of languages in the limit from positive data if and only if for all $L \in \mathcal{L}$, for all positive presentations of L, there is some point $n \in \mathbb{N}$ at which $\textsc{gim}$ converges and $L(\textsc{gim}([n])) = L$. Intuitively, given any language in $\mathcal{L}$, $\textsc{gim}$ can learn from some finite sequence of examples of strings in L a grammar that represents L. This idea of learning is very general, and there are many classes of formal languages for which such learning results exist. For reviews of some of these classes, see [7], [42]. Thus, while demonstrated with a particular subclass of the Locally k-Testable languages, the results in this paper are independent of the particular class from which the specification languages of the agents are drawn, as long as the class is identifiable in the limit from positive data.

C. Reinforcement Learning

A finite Markov decision process ($\textsc{mdp}$) M is a tuple $\{S; A; R; T; \}$ where S is finite set of states, A is finite set of available actions in each state, $R(s;) \in [0; 1]$ is the *reward* assigned to performing action in state s, $T(s; ; s')$ is the probability of transition from state s to state s' by performing action , and $\in [0; 1]$ is a *discount factor*. A *policy* is a map that assigns actions to states i.e.: $S \to A$. In other words, the policy determines which action is to be executed in each state. The *value* of state s under policy is denoted $v_M(s)$ and is defined as the expected sum of discounted rewards, when the is executed, expressed as

$$v_M(s) = E_M \left\{ R\,s; (s) + \sum_{t=1}^{\infty} {}^t R\,s_t; (s_t) \right\};$$

where the discount factor reflects the preference of immediate rewards over future ones. Similarly defined is the value of state-action pair $(s;)$ under policy :

$$Q_M(s;) = E_M \left\{ R(s;) + \sum_{t=1}^{\infty} {}^t R\,s_t; (s_t) \right\}:$$

A policy is said to be *optimal* if it maximizes the value of all states. The optimal policy is denoted and the maximum value of each state is denoted $v_M(s)$. The corresponding optimal state-action pair value is also denoted $Q_M(s;)$, and we have that $v_M(s) = \max Q_M(s;)$. One restricts the reward to take values in $[0; 1]$ without loss of generality; an equivalent $\textsc{mdp}$ with $R(s;) \in [0; 1]$ can be built from any $\textsc{mdp}$ with arbitrary reward range [43].

A $\textsc{rl}$ algorithm [25] is expected to converge to the optimal policy for an $\textsc{mdp}$, when the actual transition probabilities and/or reward function are not known. The procedure involves exploration of the $\textsc{mdp}$ model. An $\textsc{rl}$ algorithm usually maintains a table of state-action pair value estimates $Q(s;)$ that are updated based on the exploration data. Reinforcement learning algorithms have been classified as *model-based* or *model-free*. Although the characterization is debatable, the meaning of classifying an $\textsc{rl}$ algorithm as "model-based," is that T and/or R are estimated based on online observations (exploration data), and the resulting estimated model subsequently informs the computation of the the optimal policy. A model-free $\textsc{rl}$ algorithm, on the other hand, would skip the construction of an estimated $\textsc{mdp}$ model, and search directly for an optimal policy over the policy space.

The probably approximately correct ($\textsc{pac}$) analysis of $\textsc{rl}$ algorithms deals with the question of how fast an $\textsc{rl}$ algorithm converges to a near-optimal policy, relative to the number of input data points it operates on. An $\textsc{rl}$ algorithm is $\textsc{pac}$ if there exists a probabilistic bound on the number of exploration steps (where input data come from) that the algorithm can take before converging to a near-optimal policy.

Definition 1. Consider that an $\textsc{rl}$ algorithm A is executing on $\textsc{mdp}$ M. Let s_t be the visited state at time step t and A_t denotes the (non-stationary) policy that A executes at t. For a given > 0 and > 0, A is a $\textsc{pac}$ $\textsc{rl}$ algorithm if there is an $N > 0$ such that with probability at least 1 and for all but N time steps,

$$v_M^{A_t}(s_t) \quad v_M(s_t) \quad : \tag{1}$$

Equation (1) is known as the -optimality condition and N as the *sample complexity* of A, which is a function of $|S|; |A|; \frac{1}{}; \frac{1}{}; \frac{1}{1}$.

IV. Problem Formulation

Consider a system consisting of a high-level coordinator (leader), indexed 0, which is networked to $\in \mathbb{N}^+$ subordinate agents. The system operates in discrete time, with each step being completed once subordinate agents perform (synchronously) their action. The index of an agent belongs in a set K, $\{1; \ldots; \}$.

Agents model themselves and their environment in the form of finite transition systems. Each agent i has is own set (alphabet) of actions, denoted $_i$, and all the states in which the agents' environment can be at are collected in a finite set . In the rehabilitation study referred to in Section I, the humanoid robot's actions could be, for example, $\{$walk toward child, walk away from child, waive hand, make sound, push button$\}$; the wheeled robot's action set could be something like {run to child, run from child, climb ramp, make sound, flash lights}; simplified environment states may include the infant making {progress toward goal, not responding, looking at robot}. The agents' common environment imposes conditional effects upon which actions can be executed at a given environment state. The dependence between agent actions and environment states is assumed Markovian: the actions that are available to each agent depend only on the current world state, while the next environment state depends entirely on how the agents act at the current state.

The systems' acceptable behavior (system specification) is understood as a formal language $\mathcal{A}$, consisting of finite sequences of agent action profiles, which for algorithmic expediency reasons will be assumed to belong in some well-characterized family of sub-regular languages [7]. Words belonging in the specification language represent admissible system action plans. Projecting an action profile sequence, or trajectory, A on a given dimension $i \in \{1, \ldots, \}$ yields the action input sequence followed by agent i and is denoted $A\langle i \rangle$.

During normal operation, the agent controllers select an action to perform at a given world state and query the coordinator if the particular action is licensed (permissible) under the system specification. Formally speaking, an action σ is licensed, if it the concatenation of all previous actions, including σ, forms a prefix of a word in the specification language. If the action is licensed, the coordinator allows the agent's controller to execute it, and will offer the agent a reward if the action brings the system closer to satisfying its objective. If the action is not permitted, the agent controller is notified to exclude it from the set of available actions at that that state.

The goal is to design an algorithmic mechanism that enables the agents to (i) discover autonomously (with the minimal query-like input from their coordinator) how to satisfy their system specification, and (ii) reach a state where they can not only maintain normal operation in the absence of the coordinator, but also *recover* the system specification $\mathcal{A}$ that dictates the whole set of acceptable system behaviors.

V. Resilience via Learning

The general technical strategy is to employ a new, computationally efficient reinforcement learning method to address objective (i), and uniquely combine it with grammatical inference methods to achieve objective (ii). The computational architecture suggested is illustrated in Fig. 2. The figure depicts the flow of information between the distributed DQ reinforcement learning agent controllers, the GIM running on each agent, and the system's coordinator.

This paper shows that under the minimal guidance of the system coordinator (allowing actions and giving rewards), individual agents can learn optimal policies for satisfying the unknown (to them) system specification, and in the process decode (some of) the rules that constrain the system's overall acceptable behavior. Thus in the event that the coordinator goes offline, and as long as the agents' learning algorithms have sufficiently converged, the agents will be able to continue to operate on their own—assuming they can still synchronize their actions as they were doing before. In fact, the learning algorithms implemented as part of the reported solution provide convergence guarantees under reasonable assumptions on data collected. Specifically, the agent controller suggested is DAs: there are guarantees on how close its computed policy is to being optimal, as a function of the algorithm's exploration steps. The agent's GIM, on the other hand, is known to be computationally efficient (it can update its hypothesis based on the previous one and current data point in polynomial time), consistent, conservative, and strongly monotonic [44].

One of the key technical challenges addressed in this paper is the ability of the agents to reconstruct their specification language from locally learned knowledge. This is a question of significance in a number of different contexts [45], and which in the framework of the present discussion finds an affirmative answer. The mathematical proof of this latter claim is constructive. The key to developing this proof is practically in the structure of the object types defined, and in the operations between the objects in these types.

A. Formal Models for Agent Dynamics and Specifications

Consider $\in \mathbb{N}_0$ agents indexed by $i \in \{1, \ldots, \} = K$. Agent dynamics are modeled as transition systems denoted T_i.

Definition 2. A transition system is a tuple $T = \langle Q, \Sigma, \to \rangle$ with

Q	a finite set of states
Σ	a finite set of actions
$\to : Q \times \Sigma \to Q$	the transition function.

A run in T of length n is an interlaced sequence of states and actions $q_0 \sigma_1 q_1 \ldots \sigma_n q_n$ in which for every $1 < i \leq n$, $\sigma_i \in \Sigma$ such that $(q_i, \sigma, q_{i+1}) \in \to$. A *trace* is the sequence of action symbols used to generate a run; e.g., the trace associated with a run $q_0 \sigma_1 q_1 \sigma_2 q_2 \sigma_3 q_3$ is $\sigma_1 \sigma_2 \sigma_3$. Traces will also be referred to as input words. A *path* is the sequence of states encountered along a run; e.g., the path associated with run $q_0 \sigma_1 q_1 \sigma_2 q_2 \sigma_3 q_3$ is $q_0 q_1 q_2 q_3$. The collection of all runs that can be generated by a transition system is referred to as its *behavior*. In view of this, the transition system that generates every run that an agent can produce (i.e., can reproduce its behavior) is referred to as the *capacity* of agent i:

Definition 3. The *capacity* of agent i is a transition system $T_i = \langle \Gamma, \Sigma_i, \to_i \rangle$ with

Γ	a finite set of (world) states
Σ_i	a finite set of actions
$\to_i : \Gamma \times \Sigma_i \to \Gamma$	the transition function.

Symbols in Γ are understood as (world) states in transition system T_i, in other words, they express the state of the world in which the agent is operating. Agents are operating in a common workspace and are therefore assumed to share alphabet Γ.

Generally speaking, transition systems are accepting whole *families* of languages. However, once initial states $\Gamma^I \subseteq \Gamma$ and final states $\Gamma^F \subseteq \Gamma$ are marked on T, the transition system becomes an automaton $\mathcal{T}$ that accepts a specific (regular) language L. Let $\mathcal{T}_i$ be the automaton derived from T_i when all states are marked as both initial and final, i.e., $\Gamma = \Gamma^I = \Gamma^F$. In the context of this paper, the process of marking particular initial and final states is thought of as a product operation [46].

Fig. 2. Conceptual diagram of the architecture, information flow, and learning functions that enable a form of learning-based resilience in multi-agent supervisory systems.

between the transition system, and a language specification automaton $T_{L_i} = \langle G_i; G_i^I; G_i^F; \Sigma_i; \to_{L_i}\rangle$.

Definition 4. The specification of agent i is an automaton $T_{L_i} = \langle G_i; G_i^I; G_i^F; \Sigma_i; \to_{L_i}\rangle$ with

G_i	a finite set of (internal) states
$G_i^I \subseteq G_i$	a finite set of initial states
$G_i^F \subseteq G_i$	a finite set of final states
Σ_i	a finite set of actions
$\to_{L_i} : G_i \times \Sigma_i \to_{L_i} G_i$	the transition function.

Agent i behaving in a way consistent with its specification is understood as having the words (traces) of T_i belonging in L_i. The capacity of the agent, as constrained by its specification, is encoded in the product automaton $T_{C_i} = T_i \otimes T_{L_i}$, where $\otimes$ denotes the standard product operation on automata [46], referred to as the *constrained agent dynamics*.

Definition 5. The *constrained dynamics* of agent i satisfying specification T_{L_i} is an automaton

$$T_{C_i} = \langle \Pi \times G_i; \Pi \times G_i^I; \Pi \times G_i^F; \Sigma_i; \to_{C_i}\rangle \qquad (2)$$

having as components

$\Pi \times G_i$	a finite set of states
$\Pi \times G_i^I$	a finite set of initial states
$\Pi \times G_i^F$	a finite set of final states
Σ_i	a finite set of actions
$\to_{C_i} : \Pi \times G_i \times \Sigma_i \to_{C_i} \Pi \times G_i$	the transition function.[a]

[a] for $\pi, \pi' \in \Pi$, $g, g' \in G_i$, and $\sigma \in \Sigma_i$, one has $\pi \xrightarrow{\sigma}_i \pi' \wedge g \xrightarrow{\sigma}_{L_i} g' \Rightarrow (\pi; g) \xrightarrow{\sigma}_{C_i} (\pi'; g')$.

B. Formal Model for the Coordinator

The closed-loop (controlled) behavior of agent i is supposed to satisfy specification T_{L_i}. The closed-loop system, which is consistent with the specification, is T_{C_i}. However, since agents are not supposed to have knowledge of L_i (or, equivalently, T_{L_i}), the product operation yielding T_{C_i} cannot be performed locally by each agent. Instead, the agents get permission to execute their actions by a *coordinator* (Fig. 2), which can be envisioned in the form of an automaton T_0 that generates all traces of admissible action profiles (acceptable behavior) that the combined system can exhibit. Automaton T_0 can be constructed as an outcome of a special product operation in the form

$$T_0 = T_{C_1} \odot \cdots \odot T_{C_N} ;$$

which will be referred to as *synchronized product*. This operation is new in the sense that it is not identical to the standard automata product operation (cf. [46]). Here, the operation enforces synchronization on a component of the state of the factors, rather than their actions.

To define the synchronized product operation, consider first just two agent constrained dynamics T_{C_1} and T_{C_2}, that share the same space Q as the first component of their state space. Recall the standard Trim operation on automata [46], which simplifies the system by retaining only its accessible[1] and co-accessible[2] states and define the synchronized product of T_{L_1} and T_{L_2} as follows.

Definition 6. The synchronized product of automata T_{C_1} and T_{C_2} is the automaton

$$\begin{aligned} T_{C_{1\odot 2}} &\triangleq T_{C_1} \odot T_{C_2} \\ &\triangleq \mathrm{Trim}\,\big(Q \times G_1 \times G_2; Q \times G_1^I \times G_2^I; \\ &\qquad Q \times G_1^F \times G_2^F; \Sigma_1 \times \Sigma_2; \to_{C_{1\odot 2}}\big) ; \end{aligned} \qquad (3)$$

where the map

$$\to_{C_{1\odot 2}} : Q \times G_1 \times G_2 \times \Sigma_1 \times \Sigma_2 \to Q \times G_1 \times G_2$$

associates $(q; g_1; g_2)$ to $(q'; g_1'; g_2')$ given input $(\sigma_1; \sigma_2)$, an event denoted $(q; g_1; g_2) \xrightarrow{(\sigma_1;\sigma_2)}_{C_{1\odot 2}} = (q'; g_1'; g_2')$, if for $q \in Q$, $g_1 \in G_1 \ni g_1'$, $g_2 \in G_2 \ni g_2'$, $\sigma_1 \in \Sigma_1$, and $\sigma_2 \in \Sigma_2$, the two automata T_{C_1} and T_{C_2} satisfy $(q; g_1) \xrightarrow{\sigma_1}_{C_1} (q'; g_1')$ and $(q; g_2) \xrightarrow{\sigma_2}_{C_2} (q'; g_2')$, respectively.

The operation is extended inductively to more factors:

$$\begin{aligned} &T_{C_1} \odot T_{C_2} \odot T_{C_3} \odot \cdots \odot T_{C_n} \\ &\quad := (\cdots((T_{C_1} \odot T_{C_2}) \odot T_{C_3}) \odot \cdots \odot T_{C_n}) : \end{aligned}$$

With that definition in place, the automaton T_0 realizing the coordinator can be defined as follows.

[1]Accessible states are all states that are reachable from initial states.
[2]Co-accessible states are states from which there exists a path to a final state.

Definition 7. The coordinator is an automaton

$$T_0 = G_1 \quad G ; \quad G_1^I \quad G^I ; \quad G_1^F \quad G^F ; \quad 1 \quad ; !$$

with components

$G_1 \quad G$ a finite set of states
$G_1^I \quad G^I$ a finite set of initial states[a]
$G_1^F \quad G^F$ a finite set of final states[b]
1 a finite set of action profiles[c]
$! : G_1 \quad G \quad 1 \quad G_1 \quad G$ the transition function[d].

[a] $G_i^I \quad G_i$.
[b] $G_i^F \quad G_i$.
[c] An action profile is a tuple of symbols from agents' alphabets.
[d] For $i ; j \; 2$, a transition $i \; ! \; (1 ;:::;) \; j$ occurs if $(i ; k) \; 2 ! \; k$ for all $k \; 2 \; K$.

The function of the coordinator is to hold and communicate the rules of behavior for the whole system. The coordinator approves and licenses the sequence of action profiles of its subordinate agents, as they attempt to optimize their behavior through the RL algorithm described in the following section. Specifically, the coordinator collects all proposed actions from its subordinate agents, attempts to run the proposed action profile on T_0, and if it can, it allows the agents to execute them. If not, the agent with the non-compliant proposed action has to substitute it with an alternative. The coordinator does not reveal to the agents which actions are allowed at each state; it only approves the actions that can be executed, rewards those with which the system makes progress toward its objective, and notifies agents who may be repeatedly visiting a world state of whether $q \; 2 \; Q$ they have visited a different version of q in their (unknown to them) constrained dynamics T_{G_i}, depending on the unobservable $g_i \; 2 \; G_i$ component of their constrained dynamics state. As it will be illustrated shortly, the information the coordinator communicates to the agents, namely (a) the approval of actions, (b) the distinctions between world states, and (c) the reward on progress toward the system objective, are key not only for the RL to construct the optimal policy, but also for the efficient identification of the overall system behavior rules.

C. PAC Learning for Running Optimally

This section illustrates how a new sample-efficient RL algorithm named DDQ (Algorithm 1) [10] is modified appropriately here for application in the present context.

Strictly speaking, an RL algorithm like DDQ is designed to operate on MDPs, whereas the agent models of Section V-A and the languages they generate are purely deterministic. The agent dynamics are hereby expressed by transition systems, which can be thought of as MDPs where transition probabilities are one. Obvious, albeit nontrivial, extensions to probabilistic automata and stochastic languages can be contemplated; these extensions, however, remain to this point beyond the scope of this paper.

In general, DDQ integrates model-based R-max, and model-free Delayed Q-learning, while preserving the desirable features of both. The DDQ algorithm, as modified for supervisory multi-agent resilience applications, is summarized in the pseudocode of Algorithm 1. This algorithm maintains state-action value estimates of the Q matrix, initially set to their maximum possible value $v_{max} = \frac{1}{1}$. The assignment in line 36 of Algorithm 1 is termed a type-1 update, while the one appearing in line 51 is referred to as a type-2 update. A type-1 update uses the m_1 most recent experiences of a state-action pair $(s,)$ to update the pair's value, while a type-2 update is realized through a value iteration algorithm (function VI in line 47) for those state-action pairs that were experienced at least m_2 times based on the maximum likelihood estimation of transition and reward functions $\hat{T}$ and $\hat{R}$ (e.g. if $n(s; ; s^0)$ is the number of times that a transition from s to s^0 occurs by performing action in s, and $n(s;)$ is the total number of times that is performed in s, then $\hat{T}(s; ; s^0) = \frac{n(s; ; s^0)}{n(s;)}$). The result of this latter value iteration at timestep t is denoted $Q_t^{vl}(s;)$. The value iteration is set to run for $\frac{\ln(1 = (_2 (1 \quad)))}{(1 \quad)}$ iterations, with tunable parameter $_2$ which guarantees the desired accuracy on the resulting estimate (see [32, Proposition 4]). A type-1 update is successful only if the condition on line 35 of the algorithm holds. The condition is set in a way that all successful type-1 updates necessarily decrease the value estimate at least by $_1 = 3 \; _2$. Similarly, a type-2 update is successful only if the condition on line 50 of the algorithm holds. The Boolean flag learn$(s;)$ regulates type-1 updates for pair $(s;)$, allowing them to occur only if learn$(s;)$ = true . The flag is set to true initially, and is reset to true whenever either some state-action pair is updated or experienced m_2 times. The flag flips to false when no updates occur within a time window in which $(s;)$ is experienced m_1 times but the pair's subsequent attempted update fails. The DDQ algorithm is also tunable via its m_1 and m_2 parameters, and is practically reduced to Delayed Q-learning for $m_2 = 1$, and R-max for $m_1 = 1$. It can be shown [10] that by selecting appropriate values for m_1 and m_2, DDQ is not only PAC but also possesses the minimum sample complexity between R-max and Delayed Q-learning in the worst case—often, it outperforms both (see [10, Theorem 2]).

We assume that all agents independently implement DDQ algorithm in a synchronized manner to optimize their local behavior without knowing their actual constrained dynamics, under the supervision of the coordinator. The modifications that involve the interaction of each DDQ implementation with the coordinator are contained in lines 15–21 of Algorithm 1. Before an agent tries out an action that is suggested by the DDQ algorithm, it checks with the coordinator to see if a particular action is allowed at that state (lines 15–18). If an action is not allowed, the corresponding Q-value is set to 0 to make sure that it will never be chosen again there. To ensure that agent action profiles keep the system synchronized on world states, the coordinator allows one agent at a time (taking turns) to implement its greedy action choice, and forces all other agents to choose allowable actions that are compatible with the greedy agent's successor world state (lines 19–21).

Algorithm 1 The DDQ algorithm

```
Inputs: S; A; ; m1; m2; 1; 2
for all s; ; s0 do
    Q(s; )  vmax              . initialize Q values to its maximum
    U(s; )  0                 . used for attempted updates of type-1
    l(s; )  0                 . counters
    b(s; )  0                 . beginning timestep of attempted update type-1
    learn(s; )  true          . learn ags
    n(s; )  0                 . number of times(s; ) is tried
    r(s; )  0                 . accumulated reward by execution of in s
end for
t  0                          . time of the most recent successful timestep
for t = 1; 2; 3; ::: do
    let s denotes the state at time t
    choose action  = arg max 02A Q(s; 0)
    while a 2 f allowed actions of s g do
        Q(s; ) = 0
         = arg max 02A Q(s; 0)
    end while
    if not greedy turn and joint action not allowed then
        coordinator licenses a different consistent action
    end if
    observe immediate reward r and next state s0
    n(s; ) = n(s; ) + 1 and
    r(s; ) = r(s; ) + r
    if b(s; )  t then
        learn(s; )  true
    end if
    if learn(s; ) = true then
        if l(s; ) = 0 then
            b(s; )  t
        end if
        l(s; )  l(s; ) + 1
        U(s; )  U(s; ) + r +  max 0 Q(s0; 0)
        if l(s; ) = m1 then
            if Q(s; )  U(s; )=m1  2 1 then
                Q(s; )  U(s; )=m1 + 1 and t  t
                t  t
            else if b(s; ) > t then
                learn(s; )  false
            end if
            U(s; )  0 and l(s; )  0
        end if
    end if
    if n(s; ) = m2 or t = t then
        t  t
        for all (s̄; ¯) with n(s̄; ¯)  m2 do
            Qvl(s̄; ¯)  VI(Q; T̂; R̂)
        end for
        for all (s̄; ¯) do
            if Qvl(s̄; ¯)  Q(s̄; ¯) then
                Q(s̄; ¯)  Qvl(s̄; ¯)
            end if
        end for
    end if
end for
```

D. Reconstructing the Coordinator's Language

For computational expedience, the paper assumes that T_{L_i} generates a language L_i that belongs to a particular subset of Locally k-Testable class of languages (see Section III). In this subclass, each string contains a specific k-factor. In other words, if z is the required k-factor, then any string w accepted by T_{L_i} can be written as $w = uzv$ where $u; v \in$.

If $_i$ is a positive presentation of L_i, and $_i[m]$ is the sequence of the images of $1; : : : ; m$ under $_i$, then the set of factors in the string $_i(r)$ associated with $r \in f 1; : : : ; m g$ with $j (r) j \geq k$ is

$$f_k \; _i(r) = z \in {}^k_i \; j \; 9u; v \in {}_i : (r) = uzv : \quad (4)$$

The learner that identifies L_i in the limit can be compactly denoted as

$$\text{GIM} \; [m] = \bigcap_{r=1}^{m} f_k \; (r) : \quad (5)$$

The learner has converged when it has identified all k-factors that the strings in the target language are supposed to have. For example, knowing that there is only one k-factor that needs to be found in all strings of L_i, one can directly determine that the learner has converged for some $m \in \mathbb{N}$ when $j\text{GIM} \; [m] \; j = 1$.

Let a symbol vector v of length , be defined as an ordered collection of symbols arranged in a column format, where at location $i \in K$ symbol $_i \in {}_i$. To save space, v may be written in row form as: $v = (_1; _2; : : : ;)$. A concatenation of symbol vectors of the same length makes an array. The array has the same number of rows as the length of any vector in this concatenation. Every distinct vector been concatenated forms a column in this array. A vector is a (trivial) array with only one column. A row in an array is understood as a string. Thus an array can be thought of being formed, either by concatenating vectors horizontally, or by stacking (appending) strings of the same length vertically.[3]

Define the class $A_{K \; n}$ of symbol arrays with jKj rows and $2n \in \mathbb{N}_0$, columns over the set of symbols $[\; Q \; [$. More specifically, constrain $A_{K \; n}$ to contain arrays produced as (horizontal) concatenations of smaller arrays of the form $[ab]^n$, $n < 1$ where

$$a = (q_1; q_2; : : : ; q); \quad 2 \quad ; q_i \in Q$$
$$b = (_1; _2; : : : ;) \in \quad :$$

The set K will be called the *support set* of the class. The support set of a class is used to index the rows of the arrays belonging in the class. To keep track of those indices, the arrays from a particular class are annotated with the support set of this class. For example, with $K = f 1; 2; : : : ; g$, an array $A_{K \; n} \in A_{K \; n}$ is written as

$$A_K = \begin{bmatrix} _0 q_{1;0} & _{1;1} & _1 q_{1;1} & _{1;2} & \cdots & _n & _1 q_{1;n} & _1 & _{1;n} \\ _0 q_{2;0} & _{2;1} & _1 q_{2;1} & _{2;2} & \cdots & _n & _1 q_{2;n} & _1 & _{2;n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots & & \vdots \\ _0 q_{;0} & _{;1} & _1 q_{;1} & _{;2} & & _n & _1 q_{;n} & _1 & _{;n} \end{bmatrix}_K : \quad (6)$$

[3]This notation distinguishes vectors from arrays and strings; strings are (horizontal) sequences of symbols without delimiters, but when writing a vector in row format, its elements are separated with a comma and are enclosed in parentheses, while an array is denoted with square brackets.

In general, elements in K need not necessarily be consecutive integers as in the example above; it is assumed, however, that they are arranged in increasing order.

Each class $A_{K \; n}$ is assumed to contain the empty array, which is a trivial array with no columns. To ground the concept of an array $A_{K \; n}$ in the context of transition systems, assume for instance that all agents share the same state set Q and set $= Q$. Then row i of $A_{K \; n}$ may be a trace for agent T_i, while every second column is an action profile for the agents in the support set K, which has to be executed synchronously by all agents.

A run on a synchronized product is a sequence of state-action pairs of the form $(q; g_1; g_2)(\;_1; \;_2)$. We will call folding the (invertible) operation f that rearranges a tuple of this type and maps it in the form of a 2×2 array:

$$(q; g_1; g_2)(\;_1; \;_2) \overset{f}{\mapsto} \begin{bmatrix} q g_1 & _1 \\ q g_2 & _2 \end{bmatrix} :$$

The folding operation is naturally extended to runs, so that a run on $T_{C_1} \; T_{C_2}$ involving m transitions maps under f uniquely to a $2 \quad 2m$ array:

$$(q_0; g_{1;0}; g_{2;0})(\;_{1;1}; \;_{2;1}) \quad (\;_{1;m}; \;_{2;m})(q_m; g_{1;m}; g_{2;m})$$
$$\overset{f}{\mapsto} \begin{bmatrix} q_0 \, g_{1;0} & _{1;1} & q_{m-1} \, g_{1;m-1} & _{1;m} \\ q_0 \, g_{2;0} & _{2;1} & q_{m-1} \, g_{2;m-1} & _{2;m} \end{bmatrix} :$$

(The final state is dropped but can readily be recovered through $!_{C_1 \; 2}$, or equivalently from $!_{C_1}$ and $!_{C_2}$.) Thus, the sequence of action profiles is rearranged as a concatenation of column vectors, interlaced by columns that have as elements the states of the factors associated to each action.

With the aid of the folding operation, the effect of the synchronized product on the behavior of its two factors is revealed under a new light: the synchronized product is a type of parallel composition that synchronizes the two automata on their world (sub)state in . This may already be obvious from Definition 6, but the folding operation also allows an equivalent algebraic characterization of the synchronized product operation.

The synchronized product operation essentially merges a run from each automaton (where states are synchronized relative to their word substate component) into a two-row array. The mechanics of this merging operation on arrays are formalized in terms of two primitive unitary operations, one on strings and another on arrays.

A string u formed with symbols in an alphabet can be projected to ^{0}, by "deleting" all symbols in the string that do not belong in ^{0}. The projection to ^{0} operation is denoted $\$_{0} : \; ! \; ^0$, and defined inductively through the following mechanism described here for a 2-factor $u = s\,a$:

$$u \overset{0}{\mapsto} \begin{cases} ; & u = \\ \$_0(s); & u = s\,a;\; a \notin {}^0 \\ \$_0(s)a; & u = s\,a;\; a \in {}^0 \end{cases} :$$

The extraction operation on a (nonempty) symbol array $A_{K \; 1}$ over $[\,Q\,[$, is defined as a mapping $A_{K \; 1} ! \quad Q$ from symbol arrays of dimension 2 to strings of length 3.

To this end, let $: K ! f \; 1; : : : ; \; g$ be a one-to-one and onto mapping[4] for $= jKj$. Then for

$$A_{K \; 2} = \begin{bmatrix} q_{1;0} & _1 \\ \vdots & \vdots \\ q_{\;;0} & \end{bmatrix} ;$$

define

$$A_{K \; 2}\langle n_j \rangle , \quad q_{(n_j);0} \quad _{(n_j)} : \qquad (7)$$

If $A_{K \; 1} =$ (the empty string), then $A_{K \; 1}\langle j \rangle , \quad , \; \forall j \in \mathbb{N}$.

The extraction operation on array $A_{K \; n}$ is now defined recursively based on (7) as follows.

$$A_{K \; n}\langle n_j \rangle , \begin{cases} ; & A_{K \; n} = \\ ; & n_j \notin K \\ B_{K \; (n-1)}\langle n_j \rangle b\langle n_j \rangle; & A_{K \; n} = [B_{K \; (n-1)} \; b] : \end{cases}$$

Consider two array classes $A_{I \; n}$ and $A_{J \; n}$, that have the same row length $2n$, and non-intersecting support sets $I \cap J = \emptyset$. Let $: I \cup J ! f \; 1; : : : ; jI \cup Jjg$ be a monotonicity-preserving mapping, in the sense that $(i) < (j) \iff i < j$. An algebraic equivalent of the synchronized automata operation on arrays now takes the form of the merge operation . The (binary) merge operation $A_{I \; n} \quad A_{J \; n}$ yields a third array $A_{I \cup J \; n}$ which is empty if $\exists (i; j) \in I \quad J \; : \; A_{I \; n}\langle i \rangle \neq A_{J \; n}\langle j \rangle$, and otherwise satisfies

$$\begin{cases} A_{I \cup J \; n}\langle k \rangle = A_{I \; n}\langle k \rangle; & k \in I \\ A_{I \cup J \; n}\langle k \rangle = A_{J \; n}\langle k \rangle; & k \in J \end{cases} :$$

If $I \cap J \neq \emptyset$, the merge operation defaults to .

Assume now that $T_{C_1}, \ldots, T_{C}$ execute synchronously n transitions, producing the following runs

$$r_1 = \;_0 g_{1;0} \; _{1;1} \; _1 g_{1;1} \; _{1;2} \; _2 g_{1;2} \quad _{1;n} \; _n g_{1;n} \qquad (8a)$$
$$r_2 = \;_0 g_{2;0} \; _{2;1} \; _1 g_{2;1} \; _{2;2} \; _2 g_{2;2} \quad _{2;n} \; _n g_{2;n} \qquad (8b)$$
$$\vdots \qquad (8c)$$
$$r = \;_0 g_{\;;0} \; _{;1} \; _1 g_{\;;1} \; _{;2} \; _2 g_{\;;2} \quad _{;n} \; _n g_{\;;n} \; ; \qquad (8d)$$

which are reflected on run r_0 on T_0:

$$r_0 = \;_0 g_{1;0} \quad g_{\;;0} \; _{1;1} \quad _{;1} \; _1 g_{1;1} \quad g_{\;;1} \; _{1;2}$$
$$_{;2} \; _2 g_{1;2} \quad g_{\;;n-1} \; _{1;n} \quad _{;n} \; _n g_{1;n} \quad g_{\;;n} : \qquad (9)$$

The folding operation on r_0 would result in

$$f(r) = \begin{bmatrix} _0 g_{1;0} & _{1;1} & _1 g_{1;1} & _{1;2} & & _{n-1} g_{1;n-1} & _{1;n} \\ _0 g_{2;0} & _{2;1} & _1 g_{2;1} & _{2;2} & & _{n-1} g_{2;n-1} & _{2;n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ _0 g_{\;;0} & _{;1} & _1 g_{\;;1} & _{;2} & & _{n-1} g_{\;;n-1} & _{;n} \end{bmatrix} :$$

On the other hand,

$$f(r_1) = [\; _0 g_{1;0} \; _{1;1} \; _1 g_{1;1} \; _{1;2} \; _2 g_{1;2} \quad _{1;n} \;]$$
$$f(r_2) = [\; _0 g_{2;0} \; _{2;1} \; _1 g_{2;1} \; _{2;2} \; _2 g_{2;2} \quad _{2;n} \;]$$
$$\vdots$$
$$f(r\;) = [\; _0 g_{\;;0} \; _{;1} \; _1 g_{\;;1} \; _{;2} \; _2 g_{\;;2} \quad _{;n} \;]$$

[4]Recall that K may not necessarily contain consecutive integers.

and with with $K = \{1, \ldots, \}$ and being the identity for simplicity, the merging operation yields

$$f(r_1) \quad f(r_2) \qquad f(r\ )$$

$$= \begin{bmatrix} 0\ g_{1;0} & 1;1 & 1\ g_{1;1} & 1;2 & \cdots & n & 1\ g_{1;n} & 1 & 1;n \\ 0\ g_{2;0} & 2;1 & 1\ g_{2;1} & 2;2 & & n & 1\ g_{2;n} & 1 & 2;n \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots & & \vdots \\ 0\ g_{;0} & ;1 & 1\ g_{;1} & ;2 & & n & 1\ g_{;n} & 1 & ;n \end{bmatrix};$$

since $\$ \ f(r_i) = \$ \ f(r_j) = \ 0\ 1\ 2 \ \ n\ 1$ for $i, j \in K$.

The subclass of $A_{K\ n}$ in which every array A satisfies $\$_k(A_{K\ n}\langle k \rangle) \in L_k$ for a family of formal languages parameterized by $k \in M \quad K$, is denoted $A_{K\ n}\ \{L_i\}_{i \in M}$.

Assume now that every L_i belongs to a lattice class of formal languages [4]. Then a GIM can be constructed for every agent $i \in K$ to identify L_i in the limit from positive data. Each agent trace licensed by the coordinator constitutes a positive datum, and if enough[5] data are presented to GIM$_i$, the learner will converge to L_i in finite time.

Imagine a moment in time when the hypothesis (output) of every GIM has converged to its hypothesis $\hat{L}_i$ about its specification language L_i, and $\hat{L}_i \quad L_i$. Without loss of generality and for clarity of exposition reasons, we will assume that all agent specification languages L_i belong in the same language class, in which case all agents can run separate instantiations of the *same* GIM,[6] so that we can simplify notation by not differentiating between grammatical inference modules on different agents.

The question now is: can the agents, having knowledge of their own specification, reconstruct T_0 by communicating? The sequence of mathematical statements that follow provide an affirmative answer to this question.

Consider a sequence of runs like those in (8), produced by the repeated (synchronized) execution of DOQ algorithms on each agent $i \in K = \{1, \ldots, \}$, over $m \in \mathbb{N}$ different episodes. For agent $i \in K$, the sequence of runs would be denoted $\{r_i\}_{j=1}^{m}$, $\{r_i[1], r_i[2], \ldots, r_i[m]\}$. Then the GIM of agent i will be presented with the positive data presentation

$$_i = \{\$_i(r_i[1]), \$_i(r_i[2]), \ldots \$_i(r_i[m])\}\ ;$$

denoting $_i[j]$, $\$_i(r_i[j])$. The convergence assumption now translates to L GIM$(\ _i[m]) = L_i$.

At this point, and without any additional information about its teammates, agent i generically hypothesizes that the system behavior as encoded in the coordinator consists of $f^{\ 1}\ A_{K\ n}\ \{L_i\}$, i.e., runs associated with a $\quad 2n$ (with n arbitrarily large but finite) array class, where row i contains runs with traces in the hypothesized $\hat{L}_i$ (and due to $\hat{L}_i = L_i$, therefore accepted by T_{C_i}) and any other row j features elements of $(\ Q_j \quad _j)^n$ with projections onto that agree with the projection of row i.

The following lemma ensures that if two disjoint subsets of agents intersect the array classes they each hypothesize as the coordinator's language, they obtain exactly what they would have learned if they had been observing each other's actions and running a combined GIM.

[5]For the particular language subclass, a handful of positive examples generally suffice.

[6]This assumption can be directly lifted without impacting the analysis.

Lemma 1 ([9]). $A_{I[J\ \ n}\ \{L_i\}_{i \in I} \cap A_{I[J\ \ n}\ \{L_j\}_{j \in J} = A_{I[J\ \ n}\ \{L_k\}_{k \in I[J}$.

A direct next logical step is to ask whether it is the case that when all agents intersect their hypotheses obtained on which their individual learners have converged, the resulting array class would be identical to the one that a single hypothetical GIM would produce if it had access to all action profiles. The following lemma confirms that the answer to this question is affirmative.

Lemma 2 ([9]). $\bigcap_{i \in K} A_{K\ n}(L_i) = A_{K\ n}\ \{L_i\}_{i \in K}$.

In light of Lemma 2, once all agent GIM have converged, agents can communicate sharing their specification language hypotheses and reproduce the specification for the whole system. This information sharing is not particularly taxing; the languages considered are regular, which means that transmitting merely a single regular expression is sufficient to convey the specification of their target language. Any agent with knowledge of the regular expressions generated by its teammates is capable of reconstructing the combined system specification $A_{K\ n}(\{L_i\}_{i \in K})$. The theorem that follows combines Lemmas 1 and 2 and codifies the statement above.

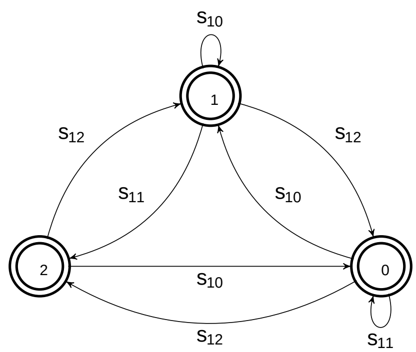
Proposition 1 ([9]). Assume that for each $i \in K$, a grammatical inference module running on input $\$_i(A[m]\langle i \rangle)$ for $A \in A$ has converged on a language L_i, for large enough $m \in \mathbb{N}$, and denote $A_{K\ n}(L_i)$ the hypothesized target language of agent i. Then $A_{K\ n}(\{L_i\}_{i \in K}) = \bigcap_{i \in K} A_{K\ n}(L_i) = A$.

E. Implementation Study

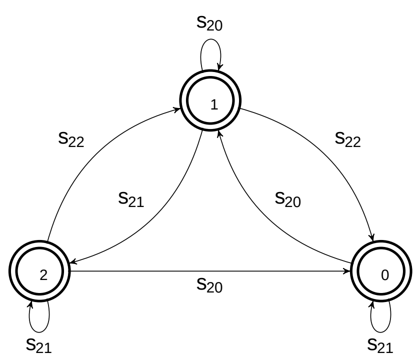
To see how the resilient learners are expected to work on a concrete example, assume that every agent specification language $L_i \ni \$_i(A_K\langle i \rangle)$ belongs to a subclass of Locally 2-Testable languages, having grammars G_i that consist of a single 2-factor, i.e., $G_i = \{\{\ _m\ _k\}\}$ for $_m, \ _k \in \ _i$. This is a very specific learning target class, chosen here for expedience of presentation—in principle, the reported approach applies to many formal language classes in the family of lattice-structured hypotheses spaces, which have been demonstrated to admit well-characterized VC-dimensions [4]. It is assumed that the class of languages in which the target specification language belongs to, is common knowledge.

The particular specification languages considered here, contain strings that include a single particular substring (2-factor) anywhere in the symbol sequence. The objective of a GIM targetting languages within this class is to identify that particular 2-factor: knowledge of this factor theoretically allows the generation of any string in the language. With knowledge that the specification language of agent i is essentially generated by a regular expression of the form $\ _m\ _k\ $ for $_m, \ _k \in \ _i$, the agent's GIM tries to infer the 2-factor $_m, \ _k$. The inference strategy is different depending on the target language class; for Locally 2-Testable languages, for instance, the learner would break each string presented to it as positive data into its 2-factors, and intersect the 2-factor sets from all presented strings to find the common elements. If a sufficiently diverse set of strings is presented to the learner, the intersection would contain only the 2-factors in the grammar G_i, thus identifying the target language; the learning algorithm would have converged. In the particular case considered here, we know that the grammar G_i has only one 2-factor, so when the aforementioned intersection has cardinality one, we know that our GIM has converged.

Consider therefore two agents, with capacities T_1 and T_2; the transition systems of the capacities of the two agents are shown in Fig. 3. The two agents share the same (discrete) world state set $= \{ _0; _1; _2\}$. Agent 1 has alphabet $_1 = \{s_{10}; s_{11}; s_{12}\}$, while agent 2 has alphabet $_2 = \{s_{20}; s_{21}; s_{22}\}$. Both agents are supervised by coordinator T_0, which determines the desired behavior of its subordinates.

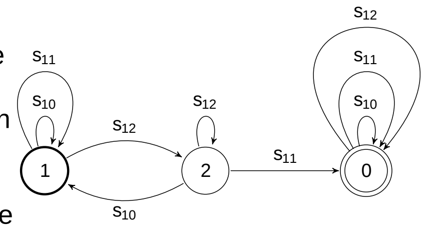


(a) T_1: capacity of agent 1

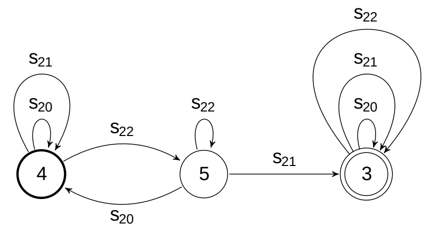


(b) T_2: capacity of agent 2

Fig. 3. The capacity of agents T_1 (a), and T_2 (b). Since in a transition system all states can be thought of as both initial and final, they are marked in the figures using circles drawn with double thick line.



(a) Specification automaton T_{L_1}



(b) Specification automaton T_{L_2}

Fig. 4. Automata T_{L_1} (a), and T_{L_2} (b), encode the specifications for agents 1 and 2, respectively. Thick single circles denote initial states; double circles denote final states. Input strings for agent 1 belong to the specification language if they contain the 2-factor $s_{12}s_{11}$. Input strings for agent 2 are consistent with that agent's specification if they contain the 2-factor $s_{22}s_{21}$.

The desired behavior for an agent is its language specification, and is encoded as an automaton: T_{L_1} for agent 1, and T_{L_2} for agent 2, and shown in Fig. 4. The labels on each specification automaton's states are (almost) arbitrary integers: the only consideration in the assignment is so that the states of the two automata can be distinguished. Here, let $G_1 = \{g_{11}; g_{12}; g_{13}\} \quad \{1; 2; 0\}$ and $G_2 = \{g_{21}; g_{22}; g_{23}\} \quad \{4; 5; 3\}$. The languages generated by T_{L_1} and T_{L_2} belong to the specific subclass of Locally 2-Testable languages considered: the specification language for agent 1 contains all strings that have $s_{12}s_{11}$ as a substring, while that for agent 2 includes all strings that have the factor $s_{22}s_{21}$.

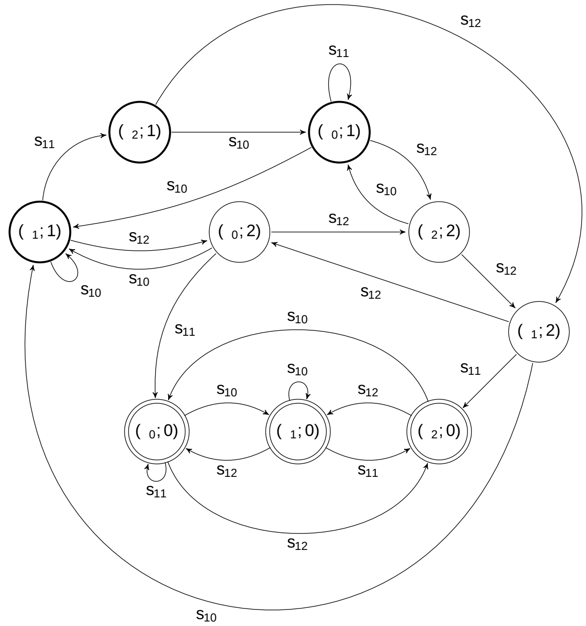
Taking the product of the agent's capacity T_i with its specification T_{L_i} produces the constrained dynamics of the agent, T_{C_i}. (Of course, neither T_{L_i} nor T_{C_i} are known to agent i.) The result of the product operation for the systems depicted in Figs. 3 and 4 is shown in Fig. 5.

It is worth noting that the product operation between the agent's capacity and its specification creates unique perspectives of a world state, from the point of view of the individual agent: for example, not only does world state $_1$ have different semantics for agent 1 compared to agent 2, for the two agents are trying to achieve different things, but there can be different instantiations of $_1$ for the same agent depending on what stage in its path to satisfying its specification the agent visits that same $_1$ state at. Since the agents are called to optimize their behavior through the DDQ algorithm without actually knowing their constrained dynamics, it is necessary for the coordinator to communicate with the agents and provide the information needed to disambiguate between their different world state instantiations. The coordinator T_0 is formed by taking the synchronized product of T_{C_1} and T_{C_2}, shown in Fig. 6.

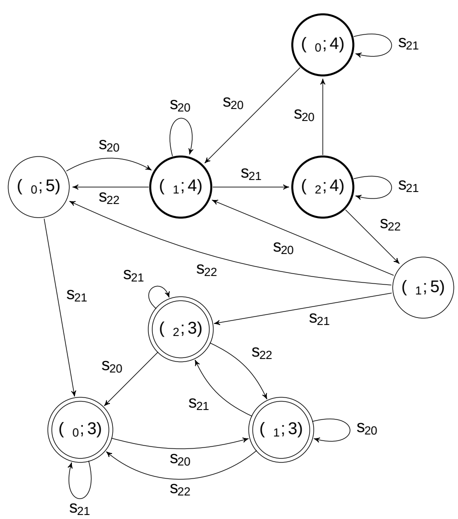
During the learning phase, when the coordinator is still operational, at each step an agent requests permission from the coordinator to implement an action, and the coordinator licenses it by offering a reward if that action contributes to satisfying the agent's specification, or rejects it if it is inconsistent with the agent's specification. Without originally knowing their specifications, agents may attempt longer sequence of actions toward their goals, but as time evolves and they learn from the rewards passed down by their coordinator, they progressively reach their goal over shorter and shorter paths.

A run in the coordinator is an action profile sequence that constitutes a plan of action for the subordinate agents. After translating tuple labels into strings (dropping parentheses and commas), this plan takes the form (9). One example is:

$$s_{\{1;2\}} = \ _1 14\ s_{10}s_{20}\ \ _1 14\ s_{12}s_{22}\ \ _0 25\ s_{11}s_{21}\ \ _0 03\ ;$$



(a) Constrained dynamics T_{C_1}



(b) Constrained dynamics T_{C_2}

Fig. 5. The constrained dynamics of agents 1 and 2. (a): $T_{C_1} = T_1 \quad T_{L_1}$; (b): $T_{C_2} = T_2 \quad T_{L_2}$.

which in tabulated form (as in (6)) looks like

$$A_{f\,1;2g} = \begin{matrix} {}_1\,14\ \ s_{10} & {}_1\,14\ \ s_{12} & {}_0\,25\ \ s_{11} \\ {}_1\,14\ \ s_{20} & {}_1\,14\ \ s_{22} & {}_0\,25\ \ s_{21} \end{matrix} \quad :$$

In practice, the action pro les are produced in a distributed fashion by the DDQ algorithms running on the agents. In one example combined execution of the learning algorithms, the very rst run attempted (one can trace it in Fig. 8) was

$$A_{f\,1;2g}(0) = \begin{matrix} {}_0\,14\ s_{11} & {}_0\,14\ s_{10} & {}_1\,14\ s_{10} & {}_1\,14\ s_{10} & {}_1\,14\ s_{11} \\ {}_0\,14\ s_{21} & {}_0\,14\ s_{20} & {}_1\,14\ s_{20} & {}_1\,14\ s_{20} & {}_1\,14\ s_{21} \end{matrix}$$
$$\begin{matrix} {}_2\,14\ s_{10} & {}_0\,14\ s_{10} & {}_1\,14\ s_{10} & {}_1\,14\ s_{12} & {}_0\,25\ s_{11} \\ {}_2\,14\ s_{20} & {}_0\,14\ s_{20} & {}_1\,14\ s_{20} & {}_1\,14\ s_{22} & {}_0\,25\ s_{21} \end{matrix} \quad : \qquad (10)$$

It should come as no surprise that the action pro les includes actions with matching indices: it is the same exact DDQ

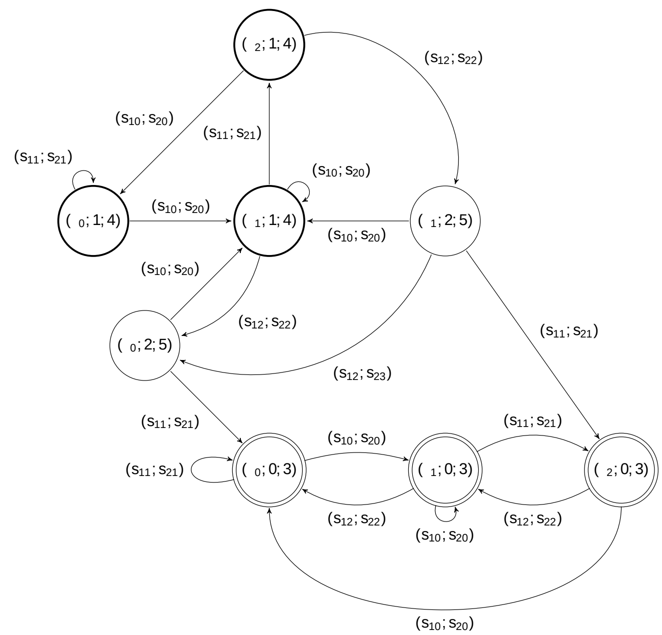


Fig. 6. The automaton T_0 of the coordinator. It is produced as $T_0 = T_{C_1} \;\; {}_2 = T_{C_1} \quad T_{C_2}$.

algorithm running on both agents, and although drawn a little differently, the automata of Figs. 5(a) and 5(b) are very similar in structure. The particular execution involved 105 accepting runs through the automaton of Fig. 6 before the DDQ algorithms converged to their optimal policies. During this learning phase, the DDQ algorithms produce positive data (each accepting trace is one positive sample) for the GIM running on each agent. It typically requires a fraction of the presentation produced by the DDQ algorithm for the GIM to identify the agent's speci cation. Different runs that agents can generate after coordinator failure are marked with different line types in Fig. 8, starting from initial states (thick single circles) and following shortest paths to nal states (double circles).

An example of this inference process is illustrated in Fig. 7. Referring back to (10), the rst data sample presented to the GIM of agent 1 was

$${}_1(0) = s_{11}\, s_{10}\, s_{10}\, s_{10}\, s_{11}\, s_{10}\, s_{10}\, s_{10}\, s_{12}\, s_{11} \quad ;$$

which can also be written as $\$ \;\; {}_1 \;\; A_{f\,1;2g}(0) h1i$.

The 2-factors of string ${}_1(0)$ are

$$f_2 \quad {}_1(0) \quad = f\, s_{11}s_{10};\ s_{10}s_{10};\ s_{10}s_{11};\ s_{10}s_{12};\ s_{12}s_{11}\, g \ ;$$

which can be seen in the leftmost box in Fig. 7. At this stage, from the viewpoint of the agent's GIM, there can be a set of possible speci cation languages, each generated by one of the following regular expressions: $s_{11}s_{10}$, $s_{10}s_{10}$, $s_{10}s_{11}$, $s_{10}s_{12}$, $s_{12}s_{11}$. There were thus ve different hypotheses for the speci cation language of agent 1. The next accepting trace that DDQ running on agent 1 produced was

$${}_1(1) = s_{11}\, s_{10}\, s_{11}\, s_{10}\, s_{11}\, s_{10}\, s_{11}\, s_{12}\, s_{12}\, s_{11} \quad ;$$

Fig. 7. The inference path that the GIM of agent 1 follows to identify the agent's specification. The GIM of the agent receives the samples in a temporal order as arranged in the graph from left to right, and whenever the new sample carries additional information about the identity of the 2-factors that generate the target language, it makes an updated hypothesis (inference). The depth in the graph of each hypothesis corresponds to the cardinality of its associated grammar G. When the grammar's cardinality reduces to one, the GIM converges. The inference path for the GIM of agent 2 is almost identical.

that generated a new set of 2-factors

$$f_2^{}{}_1(1) = \{s_{11}s_{10}, s_{10}s_{11}, s_{11}s_{12}, s_{12}s_{12}, s_{12}s_{11}\};$$

which are included in the box in Fig. 7 immediately to the right of the leftmost one. Now the set of possible specification languages is narrowed down: since the right language is one that is consistent with both the existing hypotheses (from $_1(0)$) and the new ones, the possibilities reduce to the ones generated based on the *common factors*: $\{s_{10}s_{11}, s_{11}s_{10}, s_{12}s_{11}\}$. The GIM performs an *inference* step to extract those common factors, and updates its hypothesis about the target specification language: it is one of the following: $s_{11}s_{10}$, $s_{10}s_{11}$, and $s_{12}s_{11}$.

This process is repeated with each new sample. However, the inference step is not performed on each new sample since the sample may not give GIM any additional information; in this case it merely confirms the existing hypothesis. In the case illustrated here, the GIM converged on the presentation offered by DDQ within 12 samples, at the end of which there was only one possibility left: $s_{12}s_{11}$ (see Fig. 7, bottom right).

One immediately confirms that this is indeed the specification language of agent 1 (see Fig. 4(a)). At this time, agent 1 does not need the coordinator any more to tell it what it can or cannot do; it can compute T_{L_1} and pursue the satisfaction of the system's specification independently.

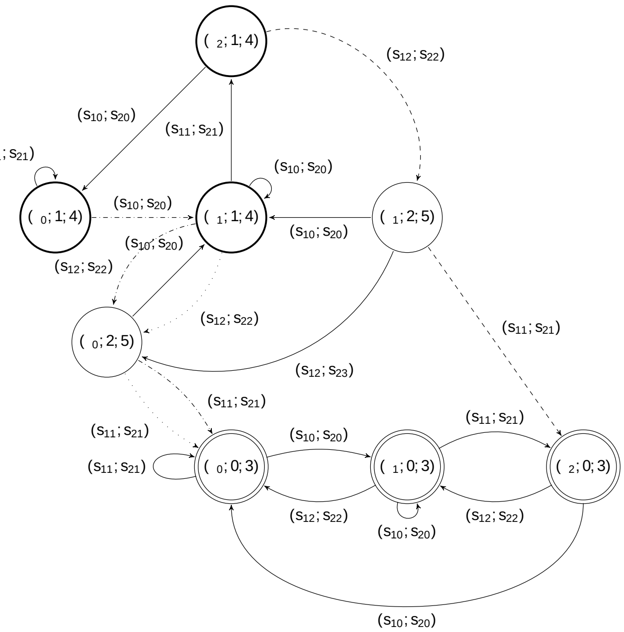


Fig. 8. The automaton T_0 of the coordinator. Lines of different type indicate the runs that the agents can produce on their own once they have learned their specification languages, after losing their coordinator. The paths they follow implement the fastest ways to satisfy their task specifications.

F. Discussion and Outlook

At first sight, the automata product operations involved in the analysis above might appear to contribute to an increase in computational complexity with the number of agents, or the size of the agents' automata. In truth, the product operations are utilized for mathematical analysis—not for system implementation. Computational complexity challenges may persist in relation to the realization of the coordinator automaton T_0. Still, at least partially, these problems can be ameliorated through a process of (automata) *factoring* [47], that circumvents the need for actually building the coordinator automaton T_0. Besides this component, in the approach of this paper (cf. [9]), learning (of both policies and specifications) is conducted in a distributed manner and should not present particular computational complexity challenges. In fact, even the sub-regular (specification) language identification realized at the level of individual agents has been formally proven to be feasible in factored form [42].

What was found to be intriguing and worthy of further investigation is the *synergy* between RL and GIM observed during our numerical implementation and testing of different scenarios within the framework of our case study. Specifically, once RL is set up to penalize unproductive (in terms of satisfying the agent's specification) actions, the DDQ algorithm will naturally gravitate to exploring shorter paths to final states. In doing so, it inevitably highlights those factors (in the case of Locally Testable languages) or subsequences (in the case of Locally Piecewise languages) that generate the unknown target language. In some sense, RL is unwittingly performing grammatical inference, and this may partially explain the rapid rate of convergence exhibited by the agents' GIM: the samples provided to them by the DDQ algorithm allowed them to hone in very quickly on the target language. This is shown more clearly in Fig. 9. The figure shows the rate of convergence of the GIM algorithm with and without DDQ. In the latter case, the GIM module is attempting to identify the specification used in Section V-E while presented with 100 positive data samples drawn uniformly from the target language. In the former case, a body of data of the same size is instead fed to the GIM algorithm directly from DDQ, as it is done in Section V-E. This comparison is repeated 50 times. The average number of grammar factors the GIM suspects as generators of the target language (we know that only one of them is correct) is plotted in Fig. 9 against the number of positive data presented so far. The figure thus indicates that when coupled with and fed by a RL algorithm, a GIM may converge up to five times faster.

Fig. 9. Convergence curves to the (single) target grammar factor in the case study of Section V-E when the GIM is driven by DDQ (red) compared to the case where the data provided to GIM are drawn uniformly from the target language (blue). Confidence areas at 95% level are drawn around the empirical averages over 50 independent runs.

While it is not anticipated that this synergy is exhibited in the process of identification in the limit of other classes of sub-regular languages, it may turn out to be a feature to be exploited in problem instantiations that combine language identification and control policy synthesis.

VI. Conclusion

Distributed multi-agent systems, in which individual agents are coordinated by a central control authority, and the dynamics of all entities is captured in the form of transition systems, can be made resilient to leader decapitation by means of learning. Specifically, under the supervision of a coordinating automaton that allows or blocks intended agent actions, local RL algorithms can optimize their agent action sequences, and in the process dramatically boost the performance of grammatical inference algorithms tasked with progressively identifying the agents' (unknown) behavior specifications. The synergy between RL and grammatical inference allows the expedient and efficient identification of the global specification, which will eventually permit the system to operate even without its coordinating algorithm. This type of result can contribute to theory that supports the design of resilient multi-agent supervisory control systems, but also be utilized from the opposite direction as a means of decoding the mechanism that generates a bundle of signals communicated over a number of different, isolated, channels. As a byproduct, the methods in this paper also hint at the possibility of developing alternative methods of performing language identification in the limit, within the classical context of reinforcement learning.

References

[1] K.-D. Kim and P. Kumar, "Cyber-physical systems: A perspective at the centennial," vol. 100, no. (Special Centennial Issue), 2012, pp. 1287–1308.

[2] S. Khaitan and J. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2015.

[3] C. Rieger, D. Gertman, and M. McQueen, "Resilient control systems: Next generation design research," in *2nd Conference on Human System Interactions*, 2009, pp. 632–636.

[4] E. Kokkoni, M. E, A. Zehfroosh, J. C. Galloway, R. Vidal, J. Heinz, and H. G. Tanner, "GEARing smart environments for pediatric motor rehabilitation," *Journal of NeuroEngineering and Rehabilitation*, vol. 17, no. 16, 2020.

[5] C. Rieger, K. Moore, and T. Baldwin, "Resilient control systems a multi-agent dynamic systems perspective," in *Proceedings of the IEEE International Conference on Electro-Information Technology*, 2013, pp. 1–13.

[6] J. M. Kendra and T. Wachtendorf, "Elements of Resilience After the World Trade Center Disaster: Reconstitung New York City's Emergency Operations Centre," *Disasters*, vol. 27, no. 1, pp. 37–53, 2003.

[7] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[8] J. Jordan, "When Heads Roll: Assessing the Effectiveness of Leadership Decapitation," *Security Studies*, vol. 18, no. 4, pp. 719–755, 2009.

[9] K. Karydis, P. Kannappan, H. G. Tanner, A. Jardine, and J. Heinz, "Resilience through learning in multi-agent cyber-physical systems," *Frontiers in Robotics and AI*, vol. 3, no. 36, 2016.

[10] A. Zehfroosh and H. G. Tanner, "A new sample-efficient PAC reinforcement learning algorithm," in *Proceedings of the 28th IEEE Mediterranean Conference on Control and Automation*, 2020, pp. 788–793.

[11] L. Guerrero-Bonilla, A. Prorok, and V. Kumar, "Formations for resilient robot teams," *IEEE Robotics and Automation Letters*, vol. 2, pp. 841–848, 2017.

[12] K. Saulnier, D. Saldana, A. Prorok, G. J. Pappas, and V. Kumar, "Resilient flocking for mobile robot teams," *IEEE Robotics and Automation Letters*, vol. 2, pp. 1039–1046, 2017.

[13] D. Saldana, A. Prorok, S. Sundaram, M. Campos, and V. Kumar, "Resilient consensus for time-varying networks of dynamic agents," in *Proceedings of the IEEE American Control Conference*, 2017, pp. 252–258.

[14] S. M. Dibaji and H. Ishii, "Resilient consensus of second-order agent networks: Asynchronous update rules over robust graphs," in *Proceedings of the IEEE American Control Conference*, 2015, pp. 1451–1456.

[15] S. M. Dibaji, H. Ishii, and R. Tempo, "Resilient randomized quantized consensus with delayed information," in *Proceedings of IEEE Conference on Decision and Control*, 2016, pp. 3505–3510.

[16] H. J. LeBlanc and X. Koutsoukos, "Resilient first-order consensus and weakly stable, higher order synchronization of continuous-time networked multiagent systems," *IEEE Transactions on Control of Network Systems*, vol. 5, pp. 1219–1231, 2018.

[17] D. Meng and K. Moore, "Studies on resilient control through multiagent consensus networks subject to disturbances," *IEEE Transactions on Cybernetics*, vol. 44, no. 11, pp. 2050–2064, 2014 .

[18] R. K. Ramachandran, P. J. A, and G. S. Sukhatme, "Resilience by reconfiguration: Exploiting heterogeneity in robot teams," *arXiv preprint arXiv: 1903.04856*, 2019.

[19] Y. Zhu, L. Zhou, Y. Zheng, J. Liu, and S. Chen, "Sampled-data based resilient consensus of heterogeneous multiagent systems," *International Journal of Robust and Nonlinear Control*, vol. 30, no. 17, pp. 1–12, 2020.