Learning Coordinated Behavior: XCSs and Statecharts

Chafic W. Bou-Saba
Elect. & Comp. Eng. Dept.
NC A&T SU
Greensboro, NC, U.S.A.
cbousaba@ncat.edu

Albert Esterline
Comp. Sci. Dept.
NC A&T SU
Greensboro, NC, U.S.A.
Esterlin@ncat.edu

Abdollah Homaifar
Elect. & Comp. Eng. Dept.
NC A&T SU
Greensboro, NC, U.S.A.
Homaifar@ncat.edu

Dan Rodgers
R&D Department
General Dynamics
Austin, TX, USA
drodgers@gdrs.com

Abstract - *We sketch a framework for learning structured coordinated behavior, specifically the tactical behavior of Experimental Unmanned Vehicles (XUVs). We conceptualize an XUV unit as a multiagent system (MAS) on which we impose a command structure to yield a holarchy, a hierarchy of holons, where a holon is both a whole and a part. The formalism used is a conservative extension of Statecharts, called a Parts/whole Statechart, which introduces a coordinating whole as a concurrent component on a par with the coordinated parts; wholes are related to common knowledge. We use X-classifier systems (XCSs), where learning acquires a population of weighted condition-action classifier rules that direct behavior. Environmental rewards modify classifier strength, and a genetic algorithm (GA) modifies the classifier population. Exploiting Statechart semantics, we translate Statechart transitions into classifiers and define data structures that interact with an XCS. Difficulties arise in learning wholes and where the GA makes structural changes.*

Keywords: *X classifier systems, Statechart semantics, Learning, Coordination, Tactical behavior, Holon*.

1 Introduction

This paper sketches a framework for learning highly structured coordinated behavior. The current application area is the tactical behavior of Experimental Unmanned Vehicles (XUVs). We conceptualize a unit of cooperating XUVs as a multiagent system (MAS) [1] since they use knowledge-level communication to accomplish cooperative tasks. The hierarchical structure of military command and control is imposed on the underlying MAS to yield a holarchy [2], a hierarchy of holons, where a holon is both a whole and a part. The formalism we use is a conservative extension of the Statechart notation, called Parts/whole Statecharts [3]. Since the implicit coordination of normal Statecharts eliminates encapsulation, a Parts/whole Statechart introduces a coordinating whole as a concurrent component on a par with the coordinated parts. We relate wholes to the technical notion of common knowledge in a group, which is a necessary condition for coordination.

The learning technique we use is X-classifier systems (XCSs) [4], where learning consists in acquiring a population of weighted condition-action classifier rules that direct behavior. A learning classifier system uses rewards from the environment to modify the strengths of the classifiers much as reinforcement learning uses rewards, but it also uses a genetic algorithm (GA) to adapt the classifier population to the learning environment. XCSs are a recent development in which the GA is applied only to the set of classifiers that advocate the selected action. By exploiting Statechart semantics, we translate Statechart transitions into classifiers and define data structures that support a simulation that interacts with an XCS. Particularly difficult aspects of this framework relate to learning wholes and to cases where the GA makes structural changes.

The remainder of this paper is organized as follows. Section 2 introduces the standard notions of MAS and holarchy as well as the reference architecture for tactical behaviors; it also presents Parts/whole Statecharts in the context of our framework. In section 3, we present classifier systems and XCS algorithmic details. Section 4 develops the framework for an XCS for acquiring a weighed population of classifier rules that reflect the structure of a holarchy formalized as a Parts/whole Statechart. Section 5 concludes and discusses future work.

2 Parts/Whole Statecharts, Holarchies

By knowledge-level communication, a group of autonomous agents forming a MAS can cooperatively accomplish complex tasks that any individual agent alone is incapable of [1]. Complex coordinated operations require social organization about which the general MAS paradigm is mute. Classically, social institutions with efficient communication and control have hierarchical structures. To analyze the coordination hierarchy of agents, we use Koestler's notion of a *holarchy* [2], which is a hierarchy of holons. A *holon* is both a whole (self-assertive) and a part (cooperative). "Holon" is a combination of the Greek word "holos," meaning whole, and the suffix "on," meaning part.

The 4D/RCS architecture [5] is a reference model for XUVs consistent with military hierarchical command doctrine. A generic control node, called an RCS_NODE, is the building block for all the hierarchical control levels. Each RCS_NODE is an identifiable part of the 4D/RCS system that has a unique identity yet is made up of sub-

ordinate parts and in turn is part of a larger whole. The RCS_NODE satisfies Koestler's definition of a holon [2], being both self-reliant and cooperative.

A holarchy is a concurrent structure, and, in general, concurrent systems are much more difficult to analyze than are sequential systems because the behaviors of components can generally be interleaved in a multitude of ways. It is thus highly desirable to use rigorous methods in specifying, designing, and testing concurrent systems. Of the various formal methods for modeling concurrent systems, automata are the most concrete and hence most accurately reflect the operations of implemented systems. The most popular automata for modeling concurrent systems are Statecharts [6], a visual formalism for the behavioral description of complex systems that extends classical state diagrams in several ways.

A Statechart lets us represent hierarchies of states and concurrently active states. A superstate that contains substates and transitions that elaborate its sequential behavior is called an XOR state since, when it is active, exactly one of its substates is active. A superstate with concurrent substates—"orthogonal components"—is called an AND state. A basic state has no substates so is neither an XOR nor an AND state. A transition has a label `e/a` indicating that, when the system is in the source state and event `e` happens, it can move to the target state on performing action `a`; the action part is optional. One substate of an XOR state is identified as the default state, the substate that becomes active when a transition is made to the XOR state. A transition may have a non-basic state as either its source or target. In fact, it may have several source states or several target states. A transition label more generally may be of the form `e[c]/a` (`a` again being optional), where `c` is a condition that must be true for the transition to fire. Frequently, the condition indicates that an orthogonal component is in a given state. The hierarchical structure of a Statechart makes it easy to suppress detail (*zoom out*) or to focus on detail (*zoom in*).

Consider the Statechart in Figure 1. The overall state, `s`, is an AND state: the system must be in both its ("orthogonal") substates, `u` and `v`. States `u` and `v` are XOR states. In state `u`, the system must be in either `m` or `k` (but not both). Similarly for states `m` (substates `n1-2`) and `v` (substates `p1-4`). An arrow starting from a darkened circle points to the default initial substate of an XOR state. The label on the transition from `k` to `m` indicates that, in state `k`, if event `a1` happens, the system can move to state `m` on performing action `b1`. Likewise for the other transitions. An action done by one orthogonal component can be a triggering event for another. E.g., when component `u` transitions from `k` to `m`, it performs action `b1`, which is the triggering event for component `v` when in state `p1` (which may then transition to state `p2`.

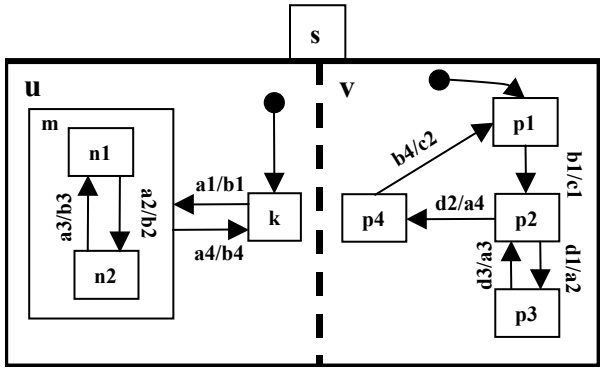

Figure 1. An XOR state with two communicating substates

The Statechart formalism is characterized not only by hierarchy (XOR states) and orthogonality (AND states) but also by broadcast communication: an action in one orthogonal component can serve as a triggering event in other components. This is one way components of a Statechart are implicitly coordinated. Others ways are by transition conditions that test the state of another component and by using the same event as a trigger in transitions in different orthogonal components. This implicit coordination introduces a web of references and eliminates encapsulation, adversely impacting the reusability, understandability, and extendibility of the resulting abstractions. We therefore follow Pazzi [3] in using an explicit approach to modeling aggregate entities, introducing a *whole* as an orthogonal component on a par with the parts that are coordinated. A part communicates only with the whole, never with other parts. The resulting *Parts/whole Statecharts* introduce no new notation but simply put constraints on how orthogonal components coordinate and require a specific section for the whole and different sections for the parts. A parts/whole Statechart naturally supports two kinds of hierarchies, XOR hierarchies and parts/whole hierarchies, the latter modeling holarchies. In fact, the Parts/whole extension of Statecharts is the perfect modeling technique to represent the holons' most fundamental characteristics, the "self-assertiveness tendency" and the "integrative tendency".

As an example, the Statechart in Figure 1 can be converted to a Part-Whole Statechart by adding (to the AND state `s`) the whole shown in Figure 2(a) and relabeling events `a2`, `a3`, and `a4` in `u` as `wa2`, `wa3`, and `wa4` and events `b1` and `b4` in `v` as `wb1` and `wb4`. Now, suppose we want to add a new part, an XOR state `r` shown in Figure 2(b). Then the whole is updated as follows. Two new states, `w11` and `w41`, say, are added. The transition labeled `b1/wb1` now has `w11` as its source (and `w2` still as its target), and the transition labeled `b4/wb4` now has `w41` as its target (and `w4` still as its source). We add a transition labeled `c2/wc2` from `w41` to `w1` and one labeled `a1/wa1` from `w1` to `w11`. Also, the event `a1` in `u` is changed to `wa1` and the event `c2` in `r` is changed to `wc2`.

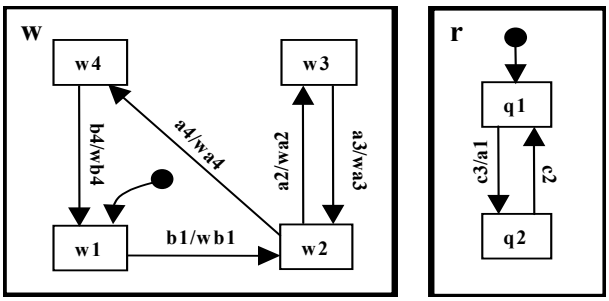

Figure 2. (a) A whole, added to Figure 1 to give a Parts/whole Statechart and (b) a new part to be added to the Parts/whole Statechart

When we model a holarchy, since the purpose of the whole is coordination, it is reasonable to consider it as corresponding to *common knowledge*, which is a necessary condition for coordination [7]. Where φ is a proposition and *G* is a group, it is common knowledge in *G* that φ if everyone in *G* knows that φ, everyone in *G* knows that everyone in *G* knows that φ, and so on for arbitrarily deep nestings of the operator "every one in *G* knows that." For example, for traffic lights to serve their purpose, it is not enough for everyone to know that red means stop and green means go. It must also be the case that everyone knows that everyone knows this. Otherwise, for example, no one could enter an intersection with confidence on a green light. The whole in the model, then, relates to behavior that rests on common knowledge.

The notion of common knowledge can be analyzed more perspicuously in terms of a fixed point solution to the following logical equivalence, where $E_G\varphi$ mean that everyone in group *G* knows that φ:

$C_G\varphi \Leftrightarrow E_G(\varphi \wedge G_G\varphi)$

We can also characterize common knowledge in terms of shared situations [8] (which embody this fixed-point equivalence) as follows. Assume that *A* and *B* are rational agents. Then we may infer common knowledge among *A* and *B* that φ if (1) *A* and *B* know that some situation σ holds, (2) σ indicates to both *A* and *B* that both *A* and *B* know that σ holds, and (3) σ indicates to both *A* and *B* that φ. Barwise [8] concludes that the fixed-point approach is the correct analysis of common knowledge, but that common knowledge generally arises via shared situations.

Implementing a part endows an individual with the capacity for certain public behavior so that it may realize the role described by the part. To avoid an infinite regress, however, wholes are not implemented apart from the individuals instantiating the parts. So the whole must be instantiated in each part. The parts must be synchronized so that their instances of the whole are always the same. Each whole instance interacts with its co-instantiated part instance and with the part instances instantiated in the other parts

3 X-Classifier Systems

A classifier system (CS) is a machine learning system that learns syntactically simple string rules, called classifiers [4]. These classifiers direct the system's performance in an active environment. A CS derives its name from its ability to learn classifying messages from the environment into general sets. A CS uses reward (feedback) from the environment to modify the strengths of the classifiers much as reinforcement learning uses the reward, but it also uses a genetic algorithm (GA) to adapt the population of classifiers to the learning environment. A CS has three major components (1) a rule and message sub-system, (2) an apportionment of credit sub-system, and (3) classifier discovery mechanisms (primarily GA). It is able to learn with incomplete information and classify the environment into hierarchies. To start a CS, one may input as many rules as possible ; or one may start from random rules then let it learn new ones and try to improve the rules by learning from experience.

X classifier systems (XCSs) [4] are a recent development of learning classifier systems (LCSs) that can solve complex learning problems. A classifier has a *condition*, usually a string of 0s, 1s, and #s (*don't care*s). When the condition matches a binary string in the current situation, the classifier may propose its *action*. Each classifier is also characterized by three parameters: the *prediction p* estimates the expected payoff if the classifier matches and its action is chosen by the system, the *prediction error* ε estimates the errors made in the predictions., and the fitness *f* denotes the classifier's fitness based on the accuracy of a its payoff prediction (instead of the prediction itself as with earlier LCSs). *Payoff* does not refer solely to the expected reward *p* but is a combination of *p* and the payoff prediction of the best possible action in the next state. However, in the case of a single-step problem payoff reduces to the reward produced by the proposed action. Note that *e* measures the error of the predictions in units of payoff.

Figure 3 portrays one learning cycle of an XCS. The XCS interacts with the environment with a set of *detectors* (which sense the current situation) and a set of *effectors* (which modify it). Unlike the traditional LCSs, an XCS has no message list. The *population* [P] consists of all current classifiers. The *match set* [M] contains all the classifiers in [P] whose conditions match the current situation. If the size of [M] does not reach a certain threshold, then the *cover operator* is invoked to add classifiers with general conditions (containing random #s) that cover strings in the current situation and randomly chosen actions not present in [M]. The *Prediction Array* is formed by calculating the fitness-weighed average of all the actions that appear in members of [M]. The action to be executed is either the one with the highest fitness-weighted average or is chosen randomly. The latter case allows the system to *explore* new

options that would never be considered if it were always guided by experience. The *action set* [A] includes all classifiers in [M] that propose the action to be executed. After the effectors process the selected action, the *reward* from the environment is used to update values for the classifiers in [A] or in the action set $[A]_{-1}$ for the previous cycle, depending on the XCS version used. *Action set subsumption* (*AS-Sub*) removes from [A] and [P] those classifiers that can be subsumed by the most general, accurate, and sufficiently experienced classifier in [A]. The GA operates on the action sets instead of the population as a whole (as with other LCSs).

There are several other parameters associated with each classifier that are mentioned in the description of one XCS learning cyce that we are about to present. The experience *exp* counts the number of times since its creation that the classifier has belonged to an action set. The time stamp *ts* denotes the time-step of the last occurrence of the GA in an action set to which this classifier belonged. The action set size *a-s* estimates the average size of the action sets this classifier has belonged to. Finally, the numerosity *num* reflects the number of micro-classifiers (ordinary classifiers) this classifier represents.

One XCS learning cycle, then, proceeds as follows :

1- Detectors generate string for current observable environment

2-
   a. Form [M] from [P] using detector string
   b. Invoke Cover if number of actions in [M] < Threshold

3- Calculate Prediction Array from [M]

4-
   a. If RANDOM > explore probability then select action at random else select best action in Prediction Array
   b. Form [A] from [M] – All rules with selected action

5-
   a. Effectors process external action
   b. Record external reward for use in next time-step (if applicable)

6- If single-step set Prediction to external reward
   Otherwise set to highest discounted Prediction Array value plus the external reward from time t – 1

7- For all rules in [A] (t or t – 1):
   a. Calculate new prediction error using Prediction
   b. Calculate new prediction using Prediction
   c. Calculate new action set size estimate
   d. Update fitness

8- If active, apply Action Set Subsump. to [A] (t)

9-
   a. Execute GA on [A] if ave time since last > Threshold
   b. If active, apply GA Subsumption to [A] (t)
   c. Insert new rules into [P] using deletion scheme

In the next section, to avoid confusion, we use *consequent* instead of *action* to denote the second part of a classifier.

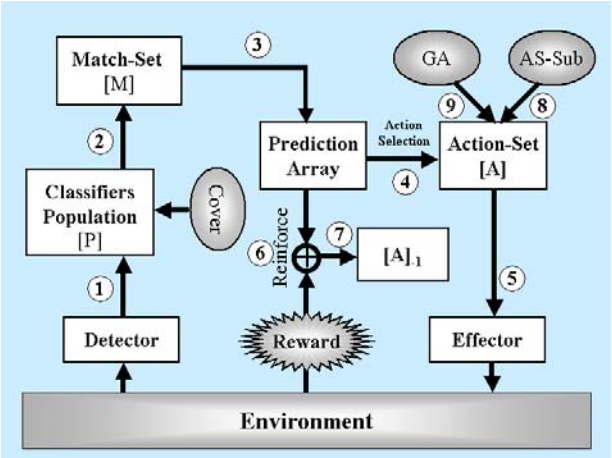

Figure 3 One learning cycle of XCS

4 Statecharts and XCS

In this section, we consider a XCS used with Statecharts restricted to their more salient features. The first subsection presents the translation of Statechart transitions to XCS classifiers. States and events or actions, however, cannot be combined arbitrarily since there are semantic connections among states, events, and actions. In subsection 4.2, we capture these relations with features and constraints on how features may be combined. Subsection 4.3 outlines the operation of the XCS excluding the use of the GA and ignoring wholes. The operation of the GA, discussed in subsection 4.4, allows for change in the structure of a Statechart, which is a complex issue because of the richness of this structure. The last subsection addresses learning wholes, which involves the difficulties of acquiring new common knowledge and the complexities of cascading structural changes to preserve coordination.

4.1 Translating Statecharts to Classifiers

To apply an XCS to a Statechart, we must translate the Statechart into classifiers with a well-defined structure. Fr simplicity, we restrict ourselves to transitions that have a single source state, a single target states, and labels of the form `e[in s]/a`, where `a` is optional. We also ignore history symbols and structural features other than XOR, AND, and basic states and transitions of form just mentioned. The basic idea, then, is that a classifier represents a transition (or a class of transitions if there are *don't care*s in the condition). One step through the XCS cycle corresponds to taking one transition in the Statechart, although the actions performed in the step must be remembered for any transitions they might subsequently trigger (all transitions in the current *micro step* in the sense of [9]). The format for a classifier, then, is

```
<source state> <event> <cond state> :
    <action> <target state>
```

Each of the five parts of this classifier is called a *role*. There are two *categories* of elements that roles denote: states and acts. The <source state>, <cond state>, and

<target state> roles denote states, while the <event> and <action> roles denote acts. The condition in this classifier, then, is the source state of the transition and the event and condition from the transition's label. The consequent is the action from the transition's label and its target state.

Each of the three roles in the condition can be a *don't care*, but a role cannot be encoded with some positions occupied by *don't care*s and some not. So no partial matching on roles is allowed. Note that a don't care in the <cond state> role corresponds to the absence of a condition in the transition label.

4.2 Representing Semantics

For applying the GA part of the XCS, it is not enough to capture just the structure. This is because the event and action on a transition generally have a semantic relation to the source and the target state. The event is something that can happen when in the source state, causing a change to the target state, and the action is something that can be done in the source state, ending that activity and initiating the activity represented by the target state. In addition, there is also a semantic relation between the event and the action: the action is an appropriate response to the event (when we are in the source state). Therefore, each state and act is associated with a feature vector. Each feature has a value in {0,1} and is represented by a unique position in the feature vector. There are consistency rules for each category, which forbid certain combinations of features. There are also agreement rules, analogous to the agreement rules in natural languages (e.g., that subject and verb must agree in person and number), which restrict the features a state or act may have in a given role in terms of the features of the elements filling the other roles.

4.3 XCS Operation

To carry out the operation of the XCS, besides the classifiers, we need to store general information about each state, which we assume is available in a *state table* indexed by state names. For each state, this records its type (the ψ function of [9]), its children (the ρ function of [9]), and its feature vector. For an XOR state, the state table also records its default state (the δ function in [9]).

The operation of the XCS is most easily pictured when the environment is a simulation that maintains a set of data structures that relate directly to Statechart semantics. The detectors then simply read from these data structures and the effectors simply write to them, and the simulation performs whatever adjustments are needed to these data structures so that they conform to Statechart semantics. To begin with, we need to record the set of acts corresponding to actions that have been performed and are still available as triggering events; we call this the *active set*. As mentioned, an act may reside in the active set for more than one XCS cycle. The active set also includes events that occur in the environment of the Statechart. We must also record the set of basic states that are currently active. This corresponds to the partial state configuration in [9] that is a component of the micro system configuration; we refer to it, however, as the *state configuration*. To be able to find the set of all active states (not just the active basic states), we need a table, the *foundation table*, that associates with each non-basic state the set of its basic descendants.

When the XCS is implemented in a group of agents corresponding to the Statechart, each agent runs an instance of the XCS procedure. Each agent is viewed as an instance of a part in the holarchy and has its own instances of the whole of which it is an immediate part and of all the wholes above it. Each agent has states that, along with the states of the other agents, contribute to the environment of the instance of the XCS procedure run by that agent. The environment itself in this sense maintains all the information that in simulation is maintained by the active set, the state configuration, and the foundation table.

In the remainder of this subsection, we focus on the case where the system is run in simulation, and we address only aspects of the XCS that are specific to Statecharts: handling non-determinism, forming the match set, what to do once a consequent is selected, and reward evaluation. The more involved issue of the operation of the GA is deferred to the following subsection and issues relating to coordination and wholes are deferred to the section after that. To begin with, then, non-determinism (as when there are two transitions with the same source state and triggering event but no distinguishing condition) is an important aspect of Statecharts. The random selection that is built into XCSs for exploration can be used here as well.

As regards forming the match set, a condition matches if (1) the state in its <source state> role is in the state configuration or has a descendant in the state configuration (as determined by the foundation table), (2) the act in its <event> role is in the active set, and (3) the state (if any) in its <cond state> role is in the state configuration or has a descendant in it.

Next, when a consequent is selected, the act in its <action> role is added to the active state. We must also remove from the state configuration the basic states that are descendants of the state filling the <source state> role. These are found by recursively querying the state table. Finally, we must add to the state configuration the basic states descended from the state in the <target state> role that become active when the corresponding transition is taken. For this, we construct a set of states, initialized to contain only the target state, by recursively querying the state table. When we come to an AND state, we remove it and add its children. When we come to an XOR state, we

replace it with its default state. The procedure terminates when only basic states remain in the set.

Finally, following [9], we take a *run* to consist of a sequence of transitions that begins with the initial configuration, where the state comprising the entire Statechart is entered. Statecharts of interest to us, unlike many, model systems expected to terminate—the mission is accomplished, aborted, or modified. So we restrict the notion of a run further by requiring that it end with a final configuration. Because of both non-determinism and the GA, there may be several alternative runs even in the same environment. Part of the reward evaluation is assigning points for the final states, but various characteristics also contribute—e.g., how long it took, resources used, risks taken, and opportunities lost. In the spirit of a holarchy, we can evaluate a part or the whole plus its parts. The overall reward for a part should include something for the performance of its unit, the larger unit of which it is a unit, and so on. How a unit's performance contributes to a part's reward, however, is a complex issue since being part of a poor unit may actually enhance our judgment of a reasonable agent (just as a player who does well on a poor team is often judged better than his statistics suggest).

4.4 GA Operation

Since the action set contains only classifiers that advocate the same consequent, crossover has an effect only on the condition parts of classifiers, although mutation can affect both the condition and the consequent of a classifier. Since the names of states and acts are largely arbitrary, and because of the semantic relations among the elements that occur in the roles in classifiers, we apply crossover and mutation to the feature vectors for these elements. We assume that there is an isomorphism between act names and act feature vectors. Thus, given an act name in a classifier, we can get its feature vector to perform crossover and mutation, and, given a feature vector that results from crossover or mutation, we can get the associated name to insert into the new classifier. In contrast, there might be several states with the same feature vector since states can be distinguished by where they occur in the Statechart.

A new classifier with only an act feature vector changed adds this transition provided the feature vector is consistent—but the transition may disrupt coordination with a whole. A straightforward way out of this problem is to record which acts are used for coordination and to block changes to such acts in classifiers. The alternative is to invoke a repair procedure that updates the whole and the other parts that coordinate with the whole.

When we have a new classifier with a state feature vector changed, the feature vector may be associated with one or more existing states, or there may be no state with that feature vector. If there is exactly one state, then a new transition to or from that state is added. If there are several states, then we can choose a state closest (in terms of the structure of the Statechart) to the other state in the classifier. If this rule does not determine a unique state, then we can turn to the feature vectors of elements filling other roles in the classifier and apply a rule based on semantic distance. If the modified state feature vector in a new classifier is associated with no existing state, we must create a new state, which generally would be in the same orthogonal component as the state filling the other state role in the new classifier. Where to place the state could again be determined in terms of structural and semantic distance.

When a new state is added, it initially occurs in only one classifier or, equivalently, takes part in only one transition (as either the source or target). If the state is a target, then there is no way out of it. If it is a source, then there is no way into it. A procedure similar to covering could be used to introduce a new, rather general classifier corresponding to multiple transitions to or from the new state.

There are many issues regarding modifications a GA may make to the structure of a Statechart. Certain changes, such as changing an XOR state to an AND state or vice versa, should clearly be forbidden. Certain large-scale changes have significant intuitive appeal. For example, moving a large coherent part of the substructure of one state to another corresponds to assigning the responsibility for part of an activity to another agent. Space restrictions prohibit exploration of such issues here.

4.5 Learning Wholes

If we consider the XCS applied to a Parts/Whole Statechart in simulation, learning coordination behaviors is apparently no more difficult than learning other behaviors except for changes caused by the GA. Such changes may modifying structure and generally require modifications to the behaviors of the parts. Accommodating these kinds of changes is similar to modifications required when goals are elaborated in some approaches to parallel planning. There has also been relevant work in game theory on communication and coordination protocols appropriate in various strategic situations [10]. In general, restructuring the coordination structure is a large problem that can be related to several relatively mature approaches.

If we consider the XCS implemented in a system modeled with a Parts/Whole Statechart, learning coordination behaviors is much more difficult since we have a copy of the whole for each group member. We can frame this problem in terms of acquiring new common knowledge, and a formal analysis reveals that this is surprisingly difficult [7]. In a controlled setting, we can generally arrange for checks, but clearly a dynamic environment over which we have little control is not good for learning coordination strategies. A particular problem

is how the group may agree on an innovation, and it is easy to show that agreement implies common knowledge [7].

5 Conclusions

We consider a group of XUVs learning military tactical behaviors, especially the coordination of units, given a formal representation of the behavior of the group. We view the control structure as a holarchy, where each agent or unit is a holon (both a whole and a part). The formalism used is an extension of the Statechart notation, called Parts/whole Statecharts [3], which introduces a coordinating whole as a concurrent component on a par with the coordinated parts. We relate wholes to the technical notion of common knowledge. The learning technique used is XCSs, where learning consists in acquiring a population of weighted condition-action classifier rules. A learning classifier system uses rewards from the environment to modify the strengths of the classifiers much as reinforcement learning uses rewards, but it also uses a genetic algorithm (GA) to adapt the classifier population to the learning environment. In XCSs, the GA is applied only to the set of classifiers advocating the selected action.

We sketched a framework for adapting XCSs to learning behavior represented by Parts/whole Statecharts. We showed how to translate a Statechart transition into a classifier. The XCS environment includes the events and states of the Statechart, represented by data structures in the simulation, and the simulation maintains the data structures in a way consistent with the Statechart semantics of [9]. States and events in a transition are not just structurally related but also semantically related. We capture the semantics with feature vectors and consistency and agreement rules. A difficult aspect arises when the GA adds states, and we must connect these states with the rest of the structure. Another difficult aspect relates to learning wholes. If we introduce new structure, this can have cascading effects if we are to maintain coordination. Also, if we consider an implemented system, learning and modifying coordination behavior amounts to acquiring new common knowledge, which is surprisingly difficult.

Two parts of our framework needing more work are learning structure and learning coordination (wholes). Hierarchy should be exploited. An attractive general approach is to zoom out to learn gross group behavior and zoom in to learn detailed behavior of its parts. Zooming in restricts the sources and targets of transitions, alleviating part of the problem with new states. Learning or accepting coordination structures is addressed by several disciplines, and we should adapt some of their approaches. Finally, there are automated translations [11] of Statecharts represented in automated tools (such as Stateflow) into other representations, some close to our manual translations. We are investigating the translations to see whether we can adapt them or create similar tools.

Acknowledgement: This work is sponsored by the US Army Research Laboratory, under contract Robotics Collaborative Technology Alliance (Contract Number DAAD19-01-2-0012). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the US Government.

References

[1] M.N. Huhns and L.M. Stephens, "Multiagent Systems and Societies of Agents," in G. Weiss (ed.), *Multiagent Systems*, MIT Press, pp. 79-120, 2000.

[2] A. Koestler, *The Ghost in the Machine*, Macmillan, New York, 1968.

[3] L. Pazzi, "Extending Statecharts for Representing Parts and Wholes," *Proc. EUROMICRO 97 Conf.*, pp. 207-214, Budapest, 1997.

[4] D. Wyatt, *Applying the XCS Learning Classifier System to Continuous-Valued Data-mining Problems*. Tech. Rep. UWELCSG05-001, Learning Classifier Sys. Group, Univ. West of England, Bristol, UK, 2004.

[5] J. Albus, *et al*., *4D/RCS: A Reference Model Architecture for Unmanned Vehicle Systems*, Version 2.0, NIST, Gaithersburg, MD, 2002.

[6] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[7] R. Fagin *et al*., *Reasoning About Knowledge*, The MIT Press, Cambridge, MA, 2003.

[8] J. Barwise, *The Situation in Logic*, CLSI Publications, Stanford, CA, 1989.

[9] D. Harel *et al*., "On the formal semantics of statecharts," *Proc. Symp. on Logic in Comp. Science* (*LICS '87*), IEEE Computer Society, pp. 54-64, 1987.

[10] M. Chwe, *Strategic Reliability of Communication Networks*, http://www.chwe.net/michael/p.pdf, 1995.

[11] E. Mikk *et al*., "Implementing Statecharts in PROMELA/SPIN," *Proc. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, FL, pp. 90-101, 1998.