Submitted, IEEE Pervasive Computing, 8 Aug 03
http://www.cds.caltech.edu/~murray/papers/2003m_km03-ieeepc.html

Distributed Computation for Cooperative Control

Eric Klavins
Electrical Engineering
University of Washington
Seattle, WA
klavins@washington.edu

Richard M. Murray
Control and Dynamical Systems
California Institute of Technology
Pasadena, CA
murray@cds.caltech.edu

Abstract

A cooperative control system consists of multiple, autonomous components interacting to control their environment. Examples include air traffic control systems, automated factories, robot soccer teams and sensor/actuator networks. Designing such systems requires a combination of tools from control theory and distributed systems. In this article, we review some of these tools and then focus on the Computation and Control Language, CCL, which we have developed as a modeling tool and a programming language for cooperative control systems.

1 Introduction

A cooperative control system consists of multiple, autonomous components interacting to control their environment. Examples include air traffic control systems, automated factories, robot soccer teams and sensor/actuator networks. In each of these systems, a component reacts to its environment and to messages received from neighboring components. Thus, a cooperative control system is at once a controlled physical system and a distributed computer. Designing cooperative control systems, therefore, requires a combination of tools from control theory and distributed systems.

A motivating example for our work is the *RoboFlag* game [3], a successor to the *RoboCup* robotic soccer tournament dedicated to the goal of building a robotic soccer team by 2050. RoboFlag is a version of "capture the flag", using the setup illustrated in Figure 1. Each team (red and blue) has a home zone, a defensive zone containing a flag and between eight and ten robots. The game can be played either with autonomous controllers or with one or two humans in the loop giving high-level directives to their teams. The goal of the red team, say, is to capture the blue team's flag and return it to the red home zone, meanwhile defending its own flag. If a red robot is "tagged" or touched by a blue robot while on the blue side of the field, it must return to its home zone for a "time out". The blue robots have a symmetric goal.

In addition to having to control their own motions, the robots have limited sensing capabilities and are organized as a distributed computational system, requiring that information be communicated between robots and across limited bandwidth links. Each robot must, therefore, contain a program that allows it to control its motion, react to events near it and participate in group strategies. Designing such programs so that they are correct, robust and fault tolerant is the goal of cooperative control.

Control vs. Distributed Computation Two very different worlds collide in cooperative control problems such as the RoboFlag game. On one hand, things like robots are electro-mechanical objects

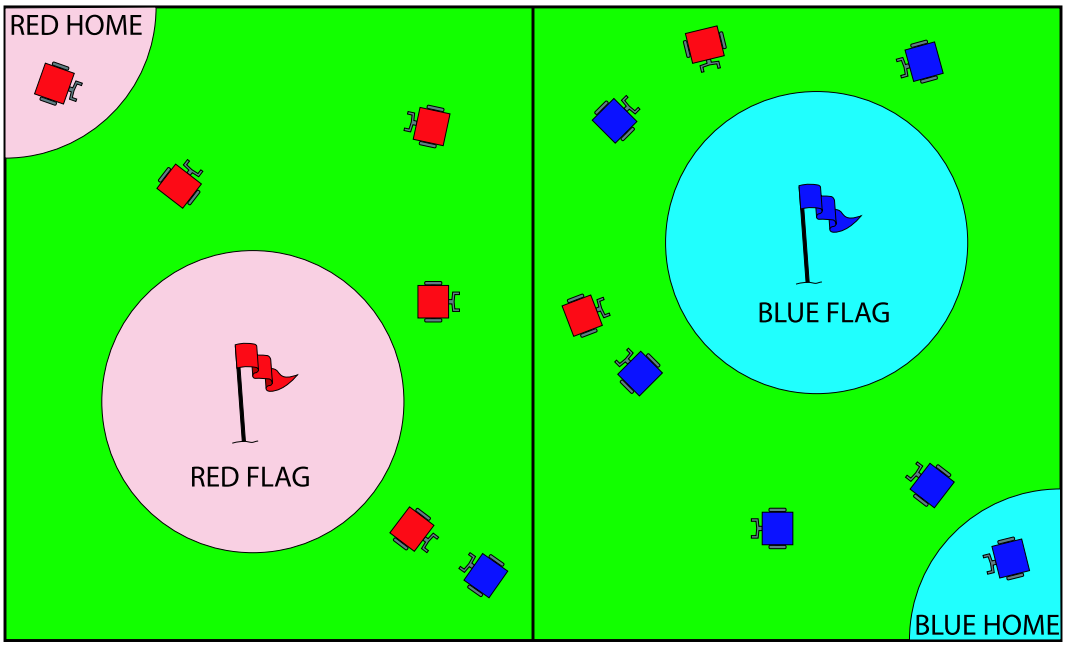


Figure 1: The *RoboFlag* game. Two teams of robots, red and blue, must defend their flags while attempting to capture the other team's flag.

whose interactions with the physical world we would like to manage. We usually speak of control with respect to a dynamical model of the world, such as a set of differential equations with inputs and outputs. The control problem is that of "closing the loop", that is, defining input rules as functions of the output values to produce a desired behavior. As control engineers, we worry about the stability, robustness and performance of the systems we design. On the other hand, a *group* of robots is by all rights a distributed computational system, each robot having its own processor and (presumably) some method of communicating with the processors on other robots. Presently, there are no universally agreed upon models for distributed systems: We might use *I/O automata*, *process algebras* or *guarded command languages* to describe how messages are passed between robots or how instructions on different processors may be interleaved. As distributed systems engineers, we worry about protocol design, deadlock avoidance and communication complexity.

The difficulty is that we cannot temporarily ignore one of these worlds while concentrating on design problems in the other. As cooperative control engineers, we must be concerned with communication protocols because they introduce delays, which are notorious for degrading performance and causing instabilities. We must mind the communication complexity of the system: A truly decentralized control algorithm will require only a few messages to be passed from robot to robot and in particular will not demand that each robot know the state of every other robot in order to act. Unfortunately, most tried and true control techniques are blind to these problems. As distributed systems engineers, we must design protocols that respect the dynamics of the environment: For example, a protocol intended to reach a consensus among a formation of aircraft about how to respond to a threat must finish before the momentum of the aircraft carries them inescapably close to the threat. In contrast, momentum and acceleration are often not a concern in traditional distributed systems wherein, for example, a bank customer can simply wait for a distributed online transaction to complete (his momentum being of no concern to the bank).

At the heart of the difference between control engineering and distributed system engineering lies the role of the environment in the design process. In control, the unpredictable, messy and incompletely understood environment is tightly coupled with a control process designed to reject disturbances from the environment to acheive a certain desired operating condition. For example, the autopilot on an aircraft will attempt to maintain altitude, heading and speed despite wind gusts and turbulance. The beauty of feedback control is that, to a large extent, it is robust to the differences in the mathematical model of how the environment affects the system and the actual effect of the environment on the system.

In contrast, in computational systems and distributed systems in particular, there often is no explicit environment whatsoever, and the notion of robustness to modeling errors is not even an issue. Such systems consist of nothing but the internal states (memories, file systems) of the processes involved. The task for the distributed systems engineer is to manipulate this information and keep it consistent among the various processes. When the issue of robustness does arise, it is with respect to whether the system can continue to function in the event that one of the processors fails.

When we design multi-vehicle systems, sensor-actuator networks or automated factories we must merge these two ways of looking at the problem. A dynamical model of the response of the system to its environment is mandatory, and so is an understanding of how information flows from process to process. We must ask questions about the stability of motions in the environment and the stability of information in the network. We must ensure that the system is robust to disturbances both physical, such as resulting from a wind gust, and logical, such as resulting from a hard reboot of a processor.

Synopsis In this article we describe at a high level some of the methods that are used to bridge these two ways of modeling and designing systems. For the sake of brevity, our review is incomplete and biased toward our own work on the *Computation and Control Language* or *CCL*, which we have begun to use for modeling control systems, especially distributed ones. To illustrate some of the concepts involved, we present a fairly complete example of a multi-robot task, inspired by the RoboFlag scenario, and show what kinds of questions we can answer about the model. Finally, we discuss one of the features of CCL, which is that it can be used as a programming language as well as a modeling tool so that the models we write down in CCL can be directly simulated or executed on hardware.

2 Models

The first step toward determining that a cooperative control system has a given property is to *write down* a description, or model, of what the system actually is to some appropriate level of detail. A control engineer might supply you with a set of differential equations that describe the closed loop (system + control) dynamics of the system. Unfortunately, once the control rules have been implemented in a distributed fashion, a simple differential equations description of the system fails to capture many important qualities, as we noted in the introduction. Thus, this description must be combined in some way with a description of the distributed system that implements the control law and that accounts for the effects of "spreading" the control law out among multiple processors.

In this section we review several formalisms for writing down (or modeling) what computation and control systems system are and what distributed systems are, starting with Hybrid Automata, then I/O Automata, temporal logic finally and UNITY. Each of these formalisms have qualities

we need for modeling Multi-Vehicle Systems, but none is entirely adequate for our purposes. Our main goal in this section, besides review, is to shed light on several important issues that led us to our present use of the *Computation and Control Language* (CCL), which we describe in the next sections.

Hybrid Automata A popular way to write down a model of system that has both continuous dynamics and discrete "modes" of control is as a *Hybrid Automaton* or *HA* [1]. HAs come in many flavors and we summarize their commonalities here. A simple finite automaton consists of a finite set of states and a set of transitions between states. For example, the level of water in a leaky water tank may be increasing *(state one)* or decreasing *(state two)*. Transitions between these two states might correspond to opening *(transition on)* or closing *(transition off)* an input valve on the tank. An HA extends the idea of a simple finite automaton with *continuous* variables that usually denote physical quantities (such as the exact level of water in the tank). An HA must say how its continuous variables change while any given state is active using a *differential inclusions* of the form

$$0 < \frac{d}{dt}h < 0.1$$

which might mean that the level of water h in a tank is increasing at a rate between 0 m/s and 0.1 m/s.

An HA assigns *guards* and *rules* to each transition. A *guard* is a predicate on the continuous state, such as $h > 5$ (read "the value of h is 5"). If the guard on a transition from state one to state two becomes true, then the discrete state changes from one to two. A *rule* is an assignment, such as $t := 0$ which might denote the reseting of a timer variable. When a transition is taken, any rules associated with it are executed.

An important aspect of HAs is that they can be *composed* to make larger models. Roughly, the composition of two HAs H_1 and H_2 is another HA denoted $H_1||H_2$ whose state set is (more or less) the cross product of the state sets of its constituents. Any transitions from H_1 and H_2 that have the same label must synchronize. For example, a water tank controller might issue *on* and *off* commands that would be synchronized with (i.e. technically they are the same transition as) the *on* and *off* transitions in the water tank model. This form of composition is acceptable for small systems. However, it is awkward for the sorts of systems found in cooperative control. For example, suppose each robot in a multi-robot system is modeled by an automaton R_i with r states. A set of n robots is modeled by $R_1||...||R_n$ which has r^n states. Furthermore, any transitions with the same label must be synchronized in the composition, which would seem to suggest that the robots are not entirely independent in this model.

I/O Automata A very successful tool for modeling distributed systems is the *I/O automaton* (IOA) model [9]. In it, an individual component is modeled as an automaton as above, except with possibly an infinite set of states. Transitions, called *actions* in IOA theory, are labeled as either *input*, *output* or *internal*. The composition of multiple components is much different than the cross product composition discussed above, however. If an IOA with output action a is composed with other IOAs, then the other IOAs must label a as an input action. An execution of an IOA consists of a sequence of actions taken by the components one at a time. A component may execute an internal action, in which case only its local state changes. A component may also execute an output action, say a, that causes all other components with (input) actions labeled by a to synchronously execute their local copies of action a, thereby changing their states. The one restriction, called a *fairness constraint*, is that each component must be allowed to take a non-input action infinitely often in

any execution. The result is an *interleaving* of actions taken by each component, with occasional partial synchronization of some of the components. In the example above, the composition of n robots in this model would have an n-dimensional vector describing its state (still living in an r^n sized space, of course, but somehow more parsimonious). Furthermore, the interleaving execution model more naturally reflects the possible ways that individual components may execute in parallel. In particular, a property P of an IOA is said to hold if and only if it holds for *all possible* fairly interleaved executions and is, therefore, robust in some sense to how the actions of the components are scheduled. IOAs have been used extensively to model distributed algorithms [9] and have proved quite amenable to analysis.

The IOA model has been extended to handle systems with continuous state variables that change according to differential equations. The result is the very comprehensive, if somewhat sophisticated, Hybrid I/O Automata (HIOA) Model [10]. In the HIOA model, continuous time variables follow trajectories according to the equations corresponding to the state of the HIOA. The trajectories are punctuated by actions taken by the various components. Because the continuous time variables of each component evolve in parallel, however, this can lead to very complex overall trajectories that are difficult to reason about.

Temporal Logic An important tool used to describe distributed systems is *temporal logic*. In temporal logic, we reason about the possible behaviors of a system (such as arising from an automaton model or a program written in Java, for example). Behaviors are defined to be sequences of states of a system. A state s, essentially a "snapshot" of a system, might assign the value of a variable x to 7 and the value of y to *true*. A behavior describes how the values of x and y change. It is important to note that there is no notion of continuous time *per se* in temporal logic, only the notion that a given state comes before or after some other state in a behavior.

Formulas in temporal logic are of the form "always P" (written $\Box P$) or "eventually Q" (or $\Diamond Q$), where P and Q are predicates on states. For example, if σ is the behavior

$$x := 1,\ x := 2,\ x := 3,\ ...,$$

assigning x to k in σ_k, then the statement $\Box x > 0$ is true of σ while the statement $\Diamond x < 0$ is false of σ. Temporal logic also defines the notion of an *action* as a relation between states. We usually write, for example, $x' = x + 1$ to denote relations between states and say that s is related to t by the action $x' = x + 1$ if the value of x in state t is equal to the value of x in state s plus one. For example, $\Box x' = x + 1$ is true of σ as defined above. Temporal logic can also be used to reason about real-time and hybrid systems with the careful use of *time* variables and yet without any further formal machinery [7].

A temporal logic formula F *specifies* a set of allowable behaviors: those behaviors for which F is true. Thus, we usually call F a specification instead of a formula. If F and G are specifications then $F \Rightarrow G$ is true if all the behaviors of F are also behaviors of G. We say that F meets the specification G. An implementation, or program, is then essentially a temporal logic formula that admits only one behavior for any initial state. Furthermore, specifications may be composed by simple conjunction. In our multi-robot example, if $R_1, ..., R_n$ are specifications of individual robot behaviors, then $R_1 \wedge ... \wedge R_n$ is their composition.

A complex temporal logic formula usually consists of two parts: a *safety* specification and a *fairness* constraint. The safefty specification is used to state what actions the components of the system are allowed to take to yield new states. The fairness constraint states when a component may take an action — infinitely often, for example. In a super simple multi-robot system, a robot

i might be described by the formula

$$\Box(x_i' = x_i + 1 \vee x_i' = x_i) \wedge \Box\Diamond(x_i' \neq x_i)$$

which states that the robot may move forward or stay still (safety) and that eventually it must move (fairness). Fairness constraints tend to get fairly complex, especially when real time is considered, and are the main source of complexity in temporal logic specifications.

In CCL, which we describe below, temporal logic is the tool we use to state the properties of the programs we write. A particularly important property is the stability of a predicate. For example, the formula

$$\Diamond\Box|x_i| < \varepsilon$$

states that x_i (a robot's position, say) is eventually always less than ε in magnitude.

UNITY The *non-duality* of programs and specifications (mentioned above) is heralded as the beauty of temporal logic and has been used with great success to reason about concurrent systems. An especially useful result of non-duality is the ease with which specifications may be automatically verified using a combination of model checking and automated theorem proving. However, a complication of the safety-fairness way of writing a specifcation is that it results in formulas that do not look very much like programs. In fact, duality may make life simpler. This is the approach taken by the *UNITY* formalism for parallel program design [2].

In UNITY, specifications S are written as a set of (possibly guarded) variable assignments. To arrive at a behavior, we simply start with some initial state, and then non-deterministically pick assignments one at a time from the set and apply them to the state to get a sequence of states. The only requirement is that each assignment is applied infinitely often in any behavior. UNITY is thus a kind of theoretical programming language that runs on an odd sort of non-deterministic machine in which a particular fairness constraint is built-in. As with CCL (described below), temporal logic turns out to be the most convenient way to reason about specifications. The general goal is to determine when a formula F is true of every behavior allowed by a specification S.

In controls, we often imagine that the components of a system are all executing their instructions at more or less the same frequency. So the fairness constraint adopted by UNITY (and IOAs) that merely states "each process gets to execute *eventually*" is somewhat too relaxed. Furthermore, writing more complicated fairness constraints in temporal logic, such as will be discussed below, can be rather cumbersome. This was a main motivation for our developing CCL, which we describe next.

3 CCL

The *Computation and Control Language*, or CCL, is a modeling language similar in appearance to UNITY, but interpreted differently. The basic unit of a CCL program is the *guarded command* (or simply command) which we describe by example. Formal definitions can be found elsewhere [6]. An example of a guarded command is:

$$t > 10 \ : \ x' \geq x + 1 \wedge t' = 0.$$

The part before the colon is called the guard and the part after it is called the rule. We interpret it as follows: If this command is executed in a state where the variable t is greater than 10, then a new state will result in which the new value of x is greater than or equal to its old value plus 1,

and the new value of t is 0. All other variables (those not occuring *primed*) remain the same. If the command is executed in a state in which t is not greater than 10, then the new state is defined to be exactly the same as the old state. The execution of a command is called a *step*. Note that guarded commands can be non-deterministic, as is the one above since it does not specify the exact new value for x, only that it should increase by at least 1.

A complete CCL program $P = (I, C)$ consists of two parts: An initial predicate I that says what the initial values of the variables involved are allowed to be; and a set C of guarded commands. Here is an example program:

Program P			
Initial	$x_1 = x_2 = 0$		
Clauses	*true*	:	$x_1' = x_1 + \delta$
	true	:	$x_2' = x_2 + \delta$

which, say, describes how the positions of two robots change. It says that initially, the robots are both at position zero and that they may move forward be δ meters upon taking a step.

CCL program composition is very straightforward. If $P_1 = (I_1, C_1)$ and $P_2 = (I_2, C_2)$, then their composition is simply $P_1 \circ P_2 = (I_1 \wedge I_2, C_1 \cup C_2)$. That is, to obtain the composition of two programs, conjoin their initial clauses and union their command sets.

A CCL program can be interpreted in various ways, depending on how its commands are scheduled for execution (or, equivalently, how we define fairness for the system). The most simple schedule is: starting with a state consistent with the initial predicate, the commands are executed in the order they were written down, over and over again. In this case, we could get something like the following execution for program P:

x_1	0	δ	δ	2δ	2δ	3δ	...
x_2	0	0	δ	δ	2δ	2δ	...

A more reasonable scheme, one that accounts for the robots not executing at the same speeds is called the *EPOCH* semantics:

> **EPOCH:** All clauses in C must be executed before any of them can be executed again[1]. A subsequence where each clause has been executed exactly once (in any order) is called an *epoch*.

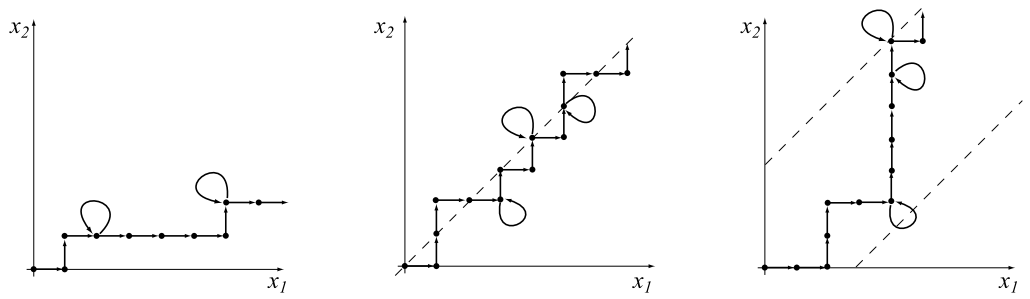
The *EPOCH* semantics allow for clauses to be executed in any order, as long as they are all "used up" before any get used again. A looser scheme is called partial synchronization, or *SYNCH(τ)* semantics, where τ is a positive integer:

> **SYNCH(τ):** For any interval of a behavior and for any two commands, the difference between the number of times each command is executed during the interval must be less than or equal to τ.

Of course, in the limit as τ approaches infinity we obtain the familiar UNITY fairness constraint: that each clause must simply be executed infinitely often. Figure 2 illustrates these different intepretations with respect to the two-robot example.

As in UNITY, we express properties of CCL programs as temporal logic formulas and define $P \models_S F$, read P *models* F with semantics S, if F is true of all behaviors allowed by program P under the interpretation S . An instructive result is the following.

[1]In all interpretations of CCL, a step may also execute no command at all, thereby leaving the state the same. This is called a *stutter step* and is useful for technical reasons beyond the scope of this article. See [8] for a discussion of stutter steps.



(a) *UNITY* (b) *EOPCH* (c) *SYNCH(3)*

Figure 2: Three different behaviors for the two-robot example. (a) A behavior allowed by the *UNITY* semantics. The difference between x_1 and x_2 may grow arbitrarily large. The loops in the behavior indicate the occurrence of one or more stutter steps. (b) A behavior allowed by the *EPOCH* semantics. After every two non-stutter steps, $x_1 = x_2$. (c) A behavior allowed by the *SYNCH(3)* semantics. The difference between x_1 and x_2 is always less than or equal 3.

Theorem 3.1 *If P is a CCL program and F is a property, then*

(i) $P \models_{SYNCH(\tau)} F \Rightarrow P \models_{SYNCH(\tau-1)} F$

(ii) $P \models_{SYNCH(2)} F \Rightarrow P \models_{EPOCH} F$

(iii) $P \models_{EPOCH} F \Rightarrow P \models_{SYNCH(1)} F$.

So if a property is true of a CCL program under a given interpretation, it is true for the more restrictive interpretation as well. This theorem along with the standard inference rules for UNITY [2], and other rules for reasoning about the more restrictive interpretations above, are the basis for reasoning about CCL programs in general [6].

Modeling Dynamical Systems: Unlike with HAs (discussed above), CCL programs do not make explicit use of continuous time. Also, a behavior should not be considered as defining a discrete time scale either. To make this clear, suppose we had a robot whose velocity is controlled by an external input. If we let x denote the position of the robot and u denote the commanded velocity, then the dynamics of the robot can be described by the differential equations

$$\frac{d}{dt}x = u.$$

This equation models the fact that the position is given by the integral of the commanded velocity. The solution to this equation for constant u is $x(t) = x(0) + ut$. To model this in CCL, we might write the program

Program P			
Initial	$x \in \mathbb{R}$		
Clauses	$true$	:	$u' = -x$
	$true$	:	$x' = x + u\delta$

where δ is a small positive constant. The first command is supposed to represent the action of the robot. It senses its location x and sets the new value of u to $-x$ (just as an example). The second

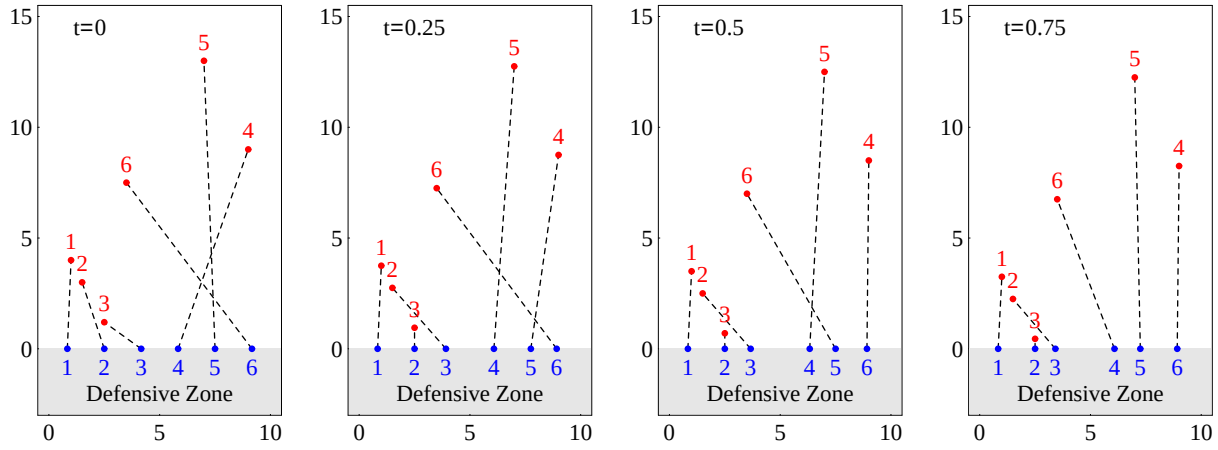


Figure 3: The first four epochs of an execution of $P_m athitrf(6)$. Dots along the x-axis represent blue defending robots. Other dots represent red attacking robots. Dashed lines represent the current assignment.

command is the action of the environment, which accounts for the actual motion of the robot. In the *EPOCH* semantics, for example, the first command may be executed and then the second, or *vice verse*, in each epoch. Thus it is possible that the robot executes its command twice in a row, which doesn't really do anything. It is as though the robot's sensor sent it the same value twice in a row, even though the environment (the robot's position) was changing. Only the execution of the second command by the environment accounts for any real passage of "time", measured here by the actual physical motion of the robot. If the second command happens to be executed twice in a row, it is as though 2δ seconds went by before the robot could again sense its position and act. The reader should try to convince his/herself that under the *SYNCH(τ)* semantics, the amount of "time" between each robot action varies between 1 and $\tau\delta$ seconds.

This treatment of time is a modeling choice on our part, and it is certianly subject to criticism. Our belief is that this is good enough for the problems we consider in which decentralized computation is as much an issue as physical dynamics, as the extended example in the next section illustrates. The conflict is between modeling continuous motion and modeling distributed systems, and CCL has proved, at least in our initial attempts, to strike a reasonable balance.

4 An Extended Example

We now reconsider the *RoboFlag* game discussed earlier. We do not propose to devise a strategy that addresses the full complexity of the game. Instead we examine the following very simple *drill* or exercise. Some number of blue robots with positions $(z_i, 0) \in \mathbb{R}^2$ must defend their zone $\{(x, y) \mid y \leq 0\}$ from an equal number of incoming red robots. The positions of the red robots are $(x_i, y_i) \in \mathbb{R}^2$. The situation is illustrated in Figure 3.

We first specify the very simplified dynamics of red robot i. It simply moves toward the defensive zone in small downward steps. When it reaches the defensive zone, it stays there (as there is no rule describing what to do if $y_i - \delta \leq 0$). The constants $min < max$ describe the boundaries of the

playing field and $\delta > 0$ is the magnitude of the distance a robot can move in one step.

Program $P_{red}(i)$			
Initial	$x_i \in [min, max] \wedge y_i > max$		
Clauses	$y_i - \delta > 0$	:	$y_i' = y_i - \delta$

The motion law for the blue team is equally simple. Each blue robot i is assigned to a red robot $\alpha(i)$. In each step, blue robot i simply moves toward the robot to which it is assigned, as long as taking such an action does not lead to a collision.

Program $P_{blue}(i)$			
Initial	$z_i \in [min, max] \ \wedge \ z_i < z_{i+1}$		
Clauses	$z_i < x_{\alpha(i)} \wedge z_i < z_{i+1} - 2\delta$	:	$z_i' = z_i + \delta$
	$z_i > x_{\alpha(i)} \wedge z_i > z_{i-1} + 2\delta$	:	$z_i' = z_i - \delta$

The dynamics of the entire drill system are defined by the composition

$$P_{drill}(n) = P_{red}(1) \circ ... \circ P_{red}(n) \circ P_{blue}(1) \circ ... \circ P_{blue}(n).$$

We now add a simple protocol for updating the assignment α. Each robot negotiates with its left and right neighbors to determine whether it should trade assignments with one of them. Switching is useful in two situations. First, if $i < j$ and $\alpha(j) < \alpha(i)$, then i and j are in conflict: intercepting their assigned red robots requires them to pass through each other. Second, if red robot $\alpha(i)$ is too close to the defensive zone for blue robot i to intercept, but not so for blue robot j, then the two robots should switch assignments. We define the predicate $switch(i, j)$ to be true if either switching the assignments of robots i and j decreases the number of red robots that can be tagged or leaves it the same and decreases the number of conflicts. The protocol is then

Program $P_{proto}(i)$			
Initial	$\alpha(i) \neq \alpha(j)$ if $i \neq j$		
Clauses	$switch(i, i+1)$	:	$(\alpha(i)', \alpha(i+1)') = (\alpha(i+1), \alpha(i))$

and the full roboflag drill system is given by

$$P_{rf}(n) = P_{drill}(n) \circ P_{proto}(1) \circ ... \circ P_{proto}(n-1).$$

Properties of P_{rf} The program we have defined has several desirable properties, which we give an overview of here. Details can be found elsewhere [6]. First, the protocol P_{rf} is *self-stabilizing* [4] in that, after an initial transient period, it settles into a mode where no assignment trades are made. That P_{rf} is self-stabilizing is expressed as

$$P_{rf}(n) \models \Diamond\Box\forall i \ \neg switch(i, i+1)$$

which states that it is eventually always true that no switches can be made. This is actually true under any fair interpretation of CCL programs. It can be proved using a *Lyapunov* style argument showing that at each step a certain non-negative quantity (essentially the number of conflicting assignments) must decrease if it is greater than zero. Self-stabilization is crucial in distributed computing. It states that no matter how the network is *perturbed* (e.g. by a process failure), it will eventually return to normal operation.

However, the duration of the transient is important. In particular, we desire that α stabilize *before* the red robots get too close to the defensive zone. Note that under a simple UNITY-like interpretation of the program, the red robots may move arbitrarily many times before the blue robots do, which is not our intention. Thus we can also show, roughly, that if the red robots are "far enough" away, then the blue robot's assignments will stabilize before they arrive at the defensive zone if, for example, the *EPOCH* interpretation is used [6]. Another property that can be shown include that the blue robots never collide (fairly evident from the guards in P_{blue}).

We have thus succeeded in formally *writing down* a complete description of a multi-vehicle task, albeit a simple one, that captures how the robots move, and how they communicate with each other to acheive their objective. Furthermore, we are able to express the properties we require of the program and reason about them.

5 Programming

Because CCL has a simple, precise and formal definition, we can easily encode it in a simple programming language, which we have done. The main benefit of programming in a language like CCL is that a CCL program bears a strong resemblance to a CCL model. In fact, they might be identical. Also, the CCL style of programming is a very natural way to write programs for control systems, where often a number of threads (here represented by CCL programs) are executed in parallel (a composition of programs).

Interested readers can obtain a version of the CCL interpreter, called CCLi, at

http://www.cs.caltech.edu/~klavins/ccl/.

The distribution consists of the intepreter; several libraries for I/O, graphics and inter/intra process communications; a good number of examples; and a user's manual. A CCL compiler is under construction. We describe some of the main features of CCLi here.

Expressions and Type Checking Basic CCLi expressions can be boolean, arithmetic, strings, lists and records. CCLi also provides lambda abstractions for defining functions (as in *Lisp* or *ML*) and also provides a simple mechanism for linking code written in other languages into CCLi. All expressions in CCLi are strongly typed and lists and lambda abstractions are polymorphic. CCLi performs *type inference* and *type checking* on programs before attempting to execute them, and gives useful error messages before exiting if your program is incorrectly typed.

Programs and Composition Programs in CCLi are very similar to how we have defined them above. Each program consists of a *name*, a list of parameters, a list of variable declarations and initializations, and a list of guarded commands. Variables are considered local to a program, unless they are "shared". Thus, CCLi defines a new kind of program composition, written as follows:

$$Q(a_1, ..., a_q) \ := \ R(b_1, ..., b_r) \ + \ S(c_1, ..., c_s) \ \ \textit{sharing} \ \ x_1, ..., x_n$$

which defines a new program Q in terms of R and S in essentially the same way as standard CCL composition, except for the "sharing" part. Any variable occuring in R but not appearing in the list $x_1, ..., x_n$ is *local* to R in Q and similarly for S. Any variable in the list $x_1, ..., x_n$ appearing in R or S is considered to be the same variable. An example CCLi program illustrating composition (although not many other features) is shown in Figure 4.

```
program plant(a,b,x0,delta) := {
  x := x0;
  y := x;
  u := 0.0;
  true : {
    x := x + delta * ( a * x + b * u ),
    y := x
  }
}

program controller ( k ) := {
  y := 0.0;
  u := 0.0;
  true : { u := - k * y }
}

program system ( x0, a, b, delta ) :=
  plant ( a, b, x0, delta ) +
  controller ( 2 * a / b ) sharing u, y;
```

Figure 4: An example CCL program defining a *plant* (the system to be controlled), a *controller*, and their closed loop combination. The *controller* program can be used to simulate the system (when composed with *plant*) or executed on actual hardware if compoed with a hardware interface program (not listed).

6 Conclusion

Cooperative control presents us with the challenge of building stable control systems in a distributed control environment. To do this, we must determine at what level we want to model the systems we build so that we can ensure they have the properties we desire for them. We argued that neither a standard control theoretic approach nor a distributed systems one is completely adequate. We also reviewed several possible formalisms that seem to be appropriate for the job, finally settling on the Computation and Control Language (CCL) which seems to capture many of the essential qualities of cooperative control systems and allows us to write succinct and natural specifications and reason about their behaviors. Finally, CCL can be used to define a programming language, CCLi, that closely mirrors the CCL formalism so that our programs and models are almost one and the same.

We have found CCL useful in other situations besides reasoning about specifications. We have, for example, used CCL to express robot communication schemes so that reasoning about their *communication complexity* is straightforward [5]. In addition we have begun to explore other control related problems such as determining the state of a communications protocol (written in CCL) based on the external movements of its participants [11].

Using formalisms like CCL to do control systems design is a young endeavor and many open problems remain. For example, a main shortcoming of discrete models like CCL is that the notion of robustness to small perturbations, seeming to require a *metric* on the state space, is not well understood, much less defined. This notion is crucial to traditional control theory. Understanding this and similar problems will enable tools from control theory and distributed computation to be used in greater harmony to build the complex control networks that promise to be ubiquitous in our future.

References

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I*, LNCS 736, pages 209–229. Springer-Verlag, 1993.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addision-Wesley, 1988.

[3] R. D'Andrea, R. M. Murray, J. A. Adams, A. T. Hayes, M. Campbell, and A. Chaudry. The RoboFlag Game. In *American Controls Conference*, 2003.

[4] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[5] E. Klavins. Communication complexity of multi-robot systems. In *Workshop on the Algorithmic Foundations of Robotics*, December 2002.

[6] E. Klavins. A formal model of a multi-robot control and communication task. In *42nd IEEE Conference on Decision and Control*, Maui, HI, December 2003.

[7] L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[8] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[9] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[10] N. A. Lynch, R. Segala, F. Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In *Hybrid Systems III*, LNCS 1066, pages 496–510. Springer-Verlag, 1996.

[11] D. Del Vecchio and E. Klavins. Observation of hybrid guarded command programs. In *42nd IEEE Conference on Decision and Control*, Maui, HI, December 2003.