Distributed Algorithms for Cooperative Control

Expressing a cooperative control system requires combining tools from control theory and distributed computation. After reviewing several possible formalisms appropriate for the job, the authors settle on the Computation and Control Language and illustrate its main features and advantages using a cooperative tracking task.

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 both to its environment and to messages received from neighboring components. A cooperative control system is at once a controlled physical system and a distributed computer, so its design must combine tools from control theory and distributed systems.

Several methods can bridge these two ways of modeling and designing systems. However, we are biased toward the Computation and Control Language, which we now use to model distributed control systems. Here, we demonstrate some of the concepts involved using a multirobot task. We also discuss CCL’s ability to be a programming language as well as a modeling tool, which lets us directly simulate or execute CCL models.

The RoboFlag example

A motivating example for our work is the RoboFlag game, a successor to the RoboCup robotic soccer tournament dedicated to 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) must defend a home zone containing a flag using between 8 and 10 robots. The game either uses autonomous robot controllers or has one or two humans in the loop giving high-level directives to their teams. The red team’s goal is to capture the blue team’s flag and return it to the red home zone while 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 in the RoboFlag example have limited sensing capabilities and are organized as a distributed computational system. This means the robots must communicate with each other, and across limited bandwidth links. Each robot must, therefore, contain a program that lets it control its motion, react to nearby events, 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, we want to manage the physical-
world interactions of electromechanical objects such as robots. We usually speak of control with respect to a dynamic world model, such as a set of differential equations with inputs and outputs. The control problem is “closing the loop”—that is, defining input rules as functions of output values to produce a desired behavior. As control engineers, we worry about our systems’ stability, robustness, and performance. 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 for communicating with other robots’ processors. Presently, no universally agreed-on models exist for distributed systems. We might use I/O automata, process algebras, or guarded command languages to describe how messages pass between robots or how to interleave instructions on different processors. As distributed systems engineers, we worry about protocol design, deadlock avoidance, and communication complexity.

Unfortunately, we can’t temporarily ignore one of these worlds while concentrating on design problems in the other. As cooperative control engineers, we must consider communication protocols because they introduce delays, which are notorious for degrading performance and causing instabilities. We must mind the system’s communication complexity. A truly decentralized control algorithm will require passing only a few messages from robot to robot and, in particular, won’t demand that each robot know every other robot’s state 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 environment’s dynamics. For example, a protocol intended to reach a consensus among an aircraft formation about how to respond to a threat must finish before the aircrafts’ momentum carries them inescapably close to the threat. In contrast, momentum and acceleration aren’t usually a concern in traditional distributed systems wherein, for example, bank customers can simply wait for distributed online transactions to complete (their momentum not being the bank’s concern).

Central to the difference between control engineering and distributed systems engineering is the environment’s role in the design process. In control engineering, the unpredictable, messy, and incompletely understood environment is tightly coupled with a control process designed to reject environmental disturbances to achieve a certain desired operating condition. For example, an aircraft’s autopilot will attempt to maintain altitude, heading, and speed despite wind gusts and turbulence. To a large extent, feedback control is robust to the difference between the mathematical model of the environment and the actual model of the environment.

In contrast, in computational systems and distributed systems in particular, there’s often no explicit environment whatsoever, and the notion of robustness to modeling errors isn’t an issue. Such systems consist only of the internal states (databases and file systems, for example) of the processes involved. The distributed systems engineer’s task is to manipulate this information and keep it consistent among the various processes. When the robustness issue does arise, it’s with respect to whether the system can continue to function if a processor fails.

When we design multivehicle systems, sensor-actuator networks, or automated factories, we must merge these two viewpoints. A dynamical model of the system’s response to its environment is mandatory, as is an understanding of how information flows from process to process. We must question 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 a gust of wind, and logical, such as the failure of a processor.

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 system’s closed-loop (system + control) dynamics. Unfortunately, once the control rules are 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. So, we must combine this description in some way with a description of the distributed system that implements the control law and accounts for the effects of “spreading” the control law out among multiple processors.

Several formalisms exist for writing down (or modeling) what computation and control systems are and what distributed systems are, such as hybrid automata, I/O automata, temporal logic, and, the Unity formalism for parallel systems. Each formalism has qualities we need for modeling multivehicle systems, but none is entirely adequate for our purposes. Consequently, several important issues led us to use CCL, which we hope our discussion of models makes clear.

Hybrid automata

A popular way to write down a system model that has both continuous dynamics and discrete control “modes” is as a hybrid automaton. HAs come in many flavors but have some commonalities. A simple finite automaton comprises a finite set of modes and a set of transitions between them. For example, the water level in a leaky water tank might be increasing (mode one) or decreasing (mode two). Transitions between these two modes 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-state variables that usually denote physical quantities (such as the exact water level in the tank). An HA must say how its continuous state changes while any given mode is active using differential inclusions of the form

\[ 0 < \frac{d}{dt} b < 0.1, \]

which might mean that a tank’s water level \( b \) 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 \( b > 3 \). If the guard on a transition from mode one to mode two becomes true, then the mode changes from one to two. A rule is an assignment, such as \( t = 0 \), which might denote resetting a timer variable. When a transition occurs, it executes any rules associated with it.

An important aspect of HAs is that you can compose them to make larger models. Roughly, the composition of two HAs, \( H_1 \) and \( H_2 \), is another HA denoted \( H_1 \parallel H_2 \), whose mode set is (more or less) the cross product of its constituents’ mode sets. Any transitions from \( H_1 \) and \( H_2 \) with the same label must synchronize. For example, a water tank controller might issue on and off commands that would be synchronized with (that is, identified with) the water tank model’s on and off transitions. This form of composition is acceptable for small systems. However, it could be awkward for the sorts of systems found in cooperative control. For example, suppose each robot in a multirobot system is modeled by an automaton \( R \), with \( r \) states. A set of \( n \) robots is modeled by \( R \parallel \ldots \parallel R_n \), having \( r^n \) states and possibly certain transitions identified so that distinguishing the individual component actions is difficult.

I/O automata

A successful tool for modeling distributed systems is the I/O automata model. In it, an individual component is modeled as an automaton, as with HAs, 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 different from the cross product composition we just discussed, however. If an IOA with output action \( a \) is composed with other IOAs, the other IOAs must label \( a \) as an input action. The execution of an IOA consists of an action sequence that the components take one at a time. A component could execute an internal action, in which case only its local state changes. A component could 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 noninput action infinitely often in any execution. The result is an interleaving of actions that each component takes, with occasional partial synchronization between communicating components.

In the robot example just mentioned, 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 can execute in parallel. In particular, an IOA property \( P \) holds if and only if it holds for all possible fairly interleaved executions and is, in a sense, robust to the order in which components’
actions are scheduled. The distributed systems community has used IOAs extensively to model distributed algorithms, and they have proved quite amenable to analysis.

Nancy Lynch and her colleagues have extended the IOA model to handle systems with continuous-state variables that change according to differential equations. The result is the comprehensive, if somewhat sophisticated, hybrid IOA model. In the HIOA model, continuous time variables follow trajectories according to the equations corresponding to the HIOA’s state. The trajectories are punctuated by actions the various components have taken. Because each component’s continuous time variables evolve in parallel, however, this can lead to complex overall trajectories about which reasoning can be difficult.

Temporal logic

An important tool used to describe distributed systems is temporal logic. In temporal logic, we reason about a system’s possible behaviors (such as those arising from an automaton model or a program written in Java). Behaviors are sequences of a system’s states. 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 x and y values change. In temporal logic, no notion of continuous time exists per se, only the notion that a given state comes before or after some other state in a behavior.

Temporal-logic formulas take the form “always P” (□P) or “eventually Q” (◇Q), where P and Q are predicates on states. For example, if σ is the behavior

\[ x := 1, \quad s := 2, \quad s := 3, \ldots, \]

assigning x to k in σ, then the statement □x > 0 is true of σ while the statement ◇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 \( \sigma \) as just defined.

We can also use temporal logic to reason about real-time and hybrid systems with the careful use of time variables but without any further formal machinery.

A temporal-logic formula \( F \) (such as \( \Box(x' = x + 1 \land x' = x_0') \land \Box \Diamond (x' \neq x_0') \)) which states that the robot can move forward or stay still (safety) and that eventually it must move (fairness). Fairness constraints tend to get fairly complex, especially when you consider real time, and they are the main source of complexity in temporal-logic specifications.

In CCL, which we describe later, we use temporal logic to state the properties of the programs we write. A particularly important property is the stability and attractivity of a predicate \( P \), written ◇□ P.

This means that from any initial condition, \( P \) is eventually always true.

A particularly important property is the stability and attractivity of a predicate \( P \), written ◇□ P. This means that from any initial condition, \( P \) is eventually always true. For example, you might specify \( P \equiv |x| < \varepsilon \). Then ◇□ P states that \( x \) (a robot’s position, say) is eventually always less than \( \varepsilon \) in magnitude, and so it expresses the property that \( x \) is ultimately bounded.

Unity

Temporal-logic practitioners have heralded the nonduality of programs and specifications (mentioned earlier) as the beauty of temporal logic. This nonduality has been used with success to model and reason about concurrent systems. An especially useful result of nonduality is the ease with which specifications can be automatically verified using a combination of model checking and automated theorem proving. However, a complication of the safety-fairness specification-writing method is that it results in formulas that don’t look much like programs. In fact, duality might simplify life, as long as the programs are written...
in a simple language. This is the approach the Unity formalism takes for parallel-program design.

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, then nondeterministically 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 nondeterministic machine in which a particular fairness constraint is built in. Because Unity (and CCL) programs essentially describe a set of possible behaviors, 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 that a specification $S$ allows.

In control engineering, we often imagine that a system’s components are all executing their instructions at more or less the same frequency. So, the fairness constraint that Unity (and IOAs) adopts that merely states “each process gets to execute eventually” is somewhat too relaxed. Furthermore, writing complicated fairness constraints (as we discuss later) in temporal logic can be cumbersome. This was a main motivation for developing CCL.

CCL

CCL is a modeling language similar in appearance to Unity but interpreted differently. A CCL program’s basic unit is the guarded command (or simply, command). You can find formal definitions elsewhere, but an example of a guarded command is

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

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 occurring primed) remain the same. If the command is executed in a state in which $t$ is not greater than 10, the new state is defined to be exactly the same as the old state. A command’s execution is called a step. Guarded commands can be nondeterministic, as is the example just presented because it doesn’t 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 involved variables’ initial values can be, and a set $C$ of guarded commands. Here is an example program that describes how the positions of two robots change:

Program $P$

<table>
<thead>
<tr>
<th>Initial</th>
<th>$x_1 = x_2 = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clauses</td>
<td>true : $x'_1 = x_1 + \delta$</td>
</tr>
<tr>
<td></td>
<td>true : $x'_2 = x_2 + \delta$</td>
</tr>
</tbody>
</table>

It says that initially, the robots are both at position zero and that they can move forward by $\delta$ meters upon taking a step.

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

We can interpret a CCL program in various ways depending on how its commands are scheduled for execution (or, equivalently, how we define system fairness). The simplest schedule is to start with a state consistent with the initial predicate and execute the commands in the order in which they are written down, over and over again. In this case, we could get something such as the following execution for program $P$:

$$x_1 \mid 0 \mid \delta \mid \delta \mid 28 \mid 28 \mid 38 \mid \ldots$$

$$x_2 \mid 0 \mid 0 \mid \delta \mid \delta \mid 28 \mid 28 \mid \ldots$$

A more reasonable scheme that would let the robots be slightly less synchronized is called Epoch fairness:

All clauses in $C$ must be executed before any of them can be executed again. A subsequence where each clause has been executed exactly once (in any order) is called an epoch.

Epoch fairness lets clauses be executed in any order as long as they’re all “used up” before any get “used” again. A looser scheme is called partial synchronization, or Synch(τ) fairness, where $\tau$ is a positive integer:

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 $\tau$.

In all interpretations of CCL, a step can also execute no command at all, leaving the state the same. This is called a stutter step and is useful for technical reasons beyond this article’s scope. Of course, in the limit as $\tau$ approaches infinity, we obtain the familiar Unity fairness constraint: each clause must simply be executed infinitely often. Figure 2 illustrates these different interpretations with respect to the two-robot example.

As in Unity, we express the properties that a CCL program might have as temporal-logic formulas and define $P \models F$, or “$P$ models $F$ with fairness constraint $S$,” to be true if $F$ is true of all behaviors the program $P$ allows under the fairness constraint $S$. An instructive result is the following theorem:

**Theorem 1.** If $P$ is a CCL program and $F$ is a property, then
1. \( P \models SYNCH(τ) \Rightarrow P \models SYNCH(τ-1) \)

2. \( P \models SYNCH(2) \Rightarrow P \models EPOCH \)

3. \( P \models EPOCH \Rightarrow P \models SYNCH(1) \)

So, if a property is true of a CCL program under a given interpretation, it’s also true for the more restrictive interpretation. This theorem, along with the standard inference rules for UNITY and other rules for reasoning about the more restrictive interpretations we mentioned, is the basis for reasoning about CCL programs in general.

Modeling dynamical systems

Unlike HA’s, CCL programs don’t explicitly use continuous time. Also, a behavior shouldn’t be considered as defining a discrete time scale. To clarify, suppose we had a robot whose velocity is controlled by an external input. If we let \( x \) denote the robot’s position and \( u \) denote the commanded velocity, we can describe the robot’s dynamics using the differential equation

\[
\frac{dx}{dt} = u.
\]

This equation models the fact that the position is given by the commanded velocity’s integral. The solution to this equation for constant \( u \) is \( x(t) = x(0) + ut \).

To model this in CCL, we might write the program

\[
\begin{array}{l}
\text{Program } P \\
\text{Initial } x \in \mathbb{R} \\
\text{Clauses true: } u' = -x \\
\text{true: } x' = x + u \delta
\end{array}
\]

where \( \delta \) is a small positive constant. The first command is supposed to represent the robot’s action. It senses its location \( x \) and sets the new value of \( u \) to \(-x\) (just as an example). The second command is the environment’s action, which accounts for the robot’s actual motion.

With EPOCH fairness, for example, the first command can be executed and then the second, or vice versa, in each epoch. So, the robot can potentially execute its command twice in a row, which doesn’t really do anything. It’s as though the robot’s sensor sent it the same value twice in a row, even though the environment’s action was changing. Only the environment’s execution of the second command accounts for any real passage of “time,” measured here by the robot’s actual physical motion. If the second command happens to be executed twice in a row, it’s as though \( 2\delta \) seconds went by before the robot could again sense its position and act. You should try to convince yourself that under SYNCH(\( \tau \)) fairness, the amount of “time” between each robot action varies between 1 and \( \tau \delta \) seconds.

Treating time this way is a modeling choice on our part, and it’s certainly subject to criticism. We believe it’s 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.

Reasoning about CCL programs

You can find more complete treatments of how we can reason about CCL programs elsewhere, but we provide a somewhat superficial description here. Given a behavior (sequence of states) \( \sigma = (s_0, s_1, \ldots) \), we write \( \sigma \models F \) to mean that the behavior \( \sigma \) satisfies the temporal-logic formula \( F \). We also define some
shorthand notation:

**Definition 1.** Let \( p \) and \( q \) be predicates. Then

1. \( p \text{ co } q \triangleq (p \Rightarrow [(q' \lor \text{skip}) \land \neg q']) \)
2. \( p \rightarrow q \triangleq (p \Rightarrow \neg q) \).

So, \( p \text{ co } q \) (\( p \) constrains \( q \)) means that whenever \( p \) is true, then the next time the state changes, \( q \) will be true. The second property, \( p \rightarrow q \) (\( p \) leads to \( q \)) means that whenever \( p \) is true, \( q \) will be true at some later time. Using these, we can state a lemma describing how to prove statements such as \( \Diamond \Box P \). Essentially, we use a Lyapunov style argument:

**Lemma 1.** Let \( F \) be a temporal-logic formula, \( V \) be a state function over the natural numbers, including zero, and \( \sigma \) be any sequence. Then

1. For any \( k \in \mathbb{N} \), if \( \sigma[V = k \Rightarrow (G \lor V < k)] \)
then \( \sigma[V = k \Rightarrow G] \).
2. For any \( k \), if \( \sigma[V = k \Rightarrow F] \) and \( \sigma[F \text{ co } F] \)
then \( \sigma[\Diamond \Box F] \).

Finally, we need tools to help us relate CCL programs to temporal-logic formulas. Several such rules exist, depending on what fairness constraints we impose. We describe a few of them here. We use the Hoare triple notation defined in CCL as follows:

**Definition 2.** Let \( a \) be an action (such as a guarded command) and let \( p \) and \( q \) be predicates. Then the Hoare triple relating \( p \) to \( q \) by \( a \) is

\[
\{p\} a \{q\} \triangleq \forall s.t. s[p] \land s[a] \Rightarrow t[q].
\]

**Lemma 2.** Let \( P = (I, C) \) be a program and let \( p \) and \( q \) be predicates. Then

1. If \( I \Rightarrow p \) and \( P \models p \land q \), then \( P \models \sqcup P \).
2. If \( \{p\} \land \neg q \) for all \( c \in C \), and if there exists an \( d \in C \) such that \( \{p\} \land \neg q \), then \( P \not\models p \land q \).

Here \( \models \) can be replaced by \( S \), where \( S \) is either \( EPOCH \) or \( SYNCH(\tau) \) for any \( \tau \).

**Lemma 3.** Let \( P = (I, C) \) be a program with \( C = \{c_1, \ldots, c_n\} \) and let \( p \) be a predicate. If there exists a predicate \( q \) such that

1. \( I \Rightarrow q \);
2. \( q \Rightarrow p \);
3. For all \( \pi \in C(1) \) it is the case that \( \{q\} c_1 \land \neg q \land c_2 \land \cdots \land c_{n-1} \land \neg q \land c_n \land \neg q \);

then \( P \models EPOCH \sqcup p \).

**Revisiting the RoboFlag example**

We now reconsider the RoboFlag game discussed earlier. We don't propose devising a strategy that addresses the game's full complexity. Instead, we examine the following simple “drill” or exercise.

Some number of blue robots with positions \( x_1, \ldots, x_n, 0 \in \mathbb{R}^2 \) must protect their defensive zone \( (x, y) \mid y \leq 0 \) from an equal number of incoming red robots. The red robots' positions are \( (x_0, y_0) \in \mathbb{R}^2 \). Figure 3 illustrates the situation.

We first specify the 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 no rule exists describing what to do if \( y_i - \delta \leq 0 \)). The constants \( \min x \) describe the playing field boundaries, and \( \delta > 0 \) is the magnitude of the distance a robot can move in one step (see Figure 4a).

The motion law for the blue team is equally simple. Each blue robot \( i \) is assigned to a red robot \( a(i) \). In each step, blue robot \( i \) simply moves toward the robot to which it is assigned, as long as taking such an action doesn't lead to a collision (see Figure 4b).

The entire drill system's dynamics are defined by the composition

\[
P_{drill}(n) = P_{red(1)} \circ \cdots \circ P_{red(n)}
\]

\[
\circ P_{blue(1)} \circ \cdots \circ P_{blue(n)}.
\]

We now add a simple protocol for updating the assignment \( a \). 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 \( a(i) = a(j) \), then \( i \) and \( j \) are in conflict. Interception their assigned red robots requires them to pass through each other. Second, if red robot \( a(i) \) 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 switching the assignments of robots \( i \) and \( j \) either decreases the number of red robots that can be tagged or leaves it the same and decreases the number of conflicts. The protocol is shown in Figure 4c, and the full RoboFlag drill system is given by

\[
P_{r}(n) = P_{drill(n)} \circ P_{proto(1)} \circ \cdots \circ P_{proto(n - 1)}.
\]

**Properties of \( P_{r} \)**

The program we've defined has several desirable properties. You can find details on how to use the tools described in the previous section to prove these properties elsewhere. First, the proto-
col $P_{rf}$ is self-stabilizing\textsuperscript{10} in that, after an initial transient period, it settles into a mode where no assignment trades are made. This is expressed as

$$P_{rf}(n) \models \text{EPOCH} \land \forall -\text{switch}(i, i+1),$$

which states that it’s eventually always true that no switches can be made. We can prove this using a few assumptions on the initial conditions and Lemma 1. The function $V$ in the lemma is essentially the number of conflicting assignments. Self-stabilization is crucial in distributed computing. It states that no matter how the network is perturbed (for example, by a process failure), it will eventually return to normal operation. However, the transient’s duration is important. In particular, we desire that $\alpha$ stabilize before the red robots get too close to the defensive zone. Under a simple Unity-like interpretation of the program, the red robots can move arbitrarily many times before the blue robots do, which isn’t our intention.

So, we can also show, using Lemma 3, that if the red robots are “far enough” away, the blue robot’s assignments will stabilize before they arrive at the defensive zone if, for example, Epoch fairness is used.\textsuperscript{9} Another demonstrated property includes 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 multivehicle task, albeit a simple one, that captures how the robots move and how they communicate with each other to achieve their objective. Furthermore, we can express the properties we require of the program and reason about them.

**Programming**

Because CCL has a simple, precise, and formal definition, we can easily encode it as a programming language, which we have done. The main benefit of programming in a language such as CCL is that a program bears a strong resemblance to a model. In fact, they might be identical. Also, the CCL programming style is a natural way to write programs.
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 a simple mechanism for linking code written in other languages into CCLi. All CCLi expressions 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 similar to how we've defined them previously. Each program comprises 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.” So, CCLi defines a new kind of program composition, written as

\[ Q(a_1, \ldots, a_n) = R(b_1, \ldots, b_m) + S(c_1, \ldots, c_p) \]

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 occurring in \( R \) but not appearing in the list \( x_1, \ldots, x_n \) is local to \( R \) in \( Q \), and similarly for \( S \). Any variable in the list \( x_1, \ldots, x_n \) appearing in \( R \) or \( S \) is considered the same variable. Figure 5 shows an example CCLi program, illustrating how we might encode the protocol part of the RoboFlag drill.

Figure 5. An example CCL program defining the protocol part of the RoboFlag drill. The functions \( f \) and \( switch \) define the conditions under which two robots would switch assignments. The \( ProtoPair \) program defines how two robots \( i \) and \( j \) interact using a single guarded command that tests \( switch \) \( i \) and \( j \) and switches assignments \( \alpha[i] \) and \( \alpha[j] \) if necessary. The last line is the \( n - 1 \) fold composition of the \( ProtoPair \) program.

W

we've 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. Additionally, we've begun to explore other control-related problems, such as determining the state of a communications protocol (written in CCL) on the basis of its participants’ external movements. We've only begun to build automated-reasoning tools for CCL, which will be based on existing tools for reasoning about temporal logic. Theoretically, a main shortcoming of discrete or hybrid formalisms such as HAs, IOAs, temporal logic, and 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 let us use tools from control theory and distributed computation in greater harmony to build the complex control networks that promise to be ubiquitous in our future.

REFERENCES


For more on this or any other computing topic, see our Digital Library at www.computer.org/publications/dlib.

**See the Future of Computing Now in IEEE Intelligent Systems**

Tomorrow’s PCs, handhelds, and Internet will use technology that exploits current research in artificial intelligence. Breakthroughs in areas such as intelligent agents, the Semantic Web, data mining, and natural language processing will revolutionize your work and leisure activities. Read about this research as it happens in *IEEE Intelligent Systems*.

**SUBSCRIBE NOW!** http://computer.org/intelligent

**Eric Klavins** is an assistant professor of electrical engineering at the University of Washington. His research interests include robotics, distributed algorithms, nonlinear systems, control, and programming languages for control systems. He received his PhD in computer science and engineering from the University of Michigan. Contact him at Electrical Eng., Univ. of Washington, Campus Box 352500, Seattle, WA 98195; klavins@ee.washington.edu.

**Richard M. Murray** is a professor of mechanical engineering and the chair of the Division of Engineering and Applied Science at the California Institute of Technology, Pasadena. His research interests include networked control systems, nonlinear control of mechanical systems, and nonlinear dynamical systems theory. He received his PhD in electrical engineering and computer science from the University of California, Berkeley. He is a member of the IEEE and the Society for Industrial and Applied Mathematics. Contact him at Control and Dynamical Systems 107-81, California Inst. of Technology, 1200 E. California Blvd., Pasadena, CA 91125; murray@cds.caltech.edu.