A Computation and Control Language for Multi-Vehicle Systems

Eric Klavins

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

***Abstract*— We introduce the Computation and Control Language (CCL), a *guarded-command* formalism for expressing systems wherein control and computation are intertwined. A CCL program consists of a set of guarded commands that update continuous or discrete variables. CCL programs are used to specify systems and program them. Their properties can be analyzed using temporal logic. In this paper, a robot capture-the-flag system, called the "RoboFlag Drill", is encoded in CCL and certain desirable properties of it are verified. The example consists of a self-stabilizing communications protocol whose behavior depends on the actions taken by the robots in their environment. The paper concludes with a brief overview of our initial implementation of the formal semantics of CCL as a practical programming language.**

I. Introduction

We are interested in designing large scale decentralized systems that have multiple computational and controlled elements, like multi-vehicle systems, sensor-actuator networks or automated factories. Generally, to understand and control these systems, a combination of control, computation and communication theory is needed at various different levels. In any non-trivial system, these seemingly separate aspects of system design become intermingled. The result is that, for example, the verification of the control part of the design *depends* heavily on the design of the communications part, requiring, ideally, that these two aspects of the design inhabit the same formalism.

For instance, the problem we consider in this paper (Section III-B) involves a group of robots that must defend their flag against a group of opponent robots in a game similar to capture-the-flag. Thus, each robot must perform a low level motion control task (tracking an opponent) and simultaneously take part in a high level communications protocol with the other robots (to decide who should track what opponent). The protocol we propose is self-stabilizing [7] under certain circumstances (explored in Section V-D) that depend on the motions of the robots. The stable configuration of the protocol corresponds to a reasonable agreement among the defending robots about who will track whom and is important for the entire control scheme to function correctly.

We specify the capture-the-flag system in a formalism called the *Computation and Control Language* (CCL). CCL is inspired by the UNITY formalism for parallel program design [3] but is adapted to dynamical systems and control tasks. Simply put, a CCL program consists of a set of guarded commands that are executed in some order at each step in the evolution of the system. One feature of CCL is that the dynamics of the environment are modeled by a subset of these commands and not by separate differential equations. The result is that the distinction between the internal (logical) and external (physical) states of the system under consideration is lost, allowing a single set of tools (standard temporal logic) to be used for modeling, specification and analysis.

The main contributions of the paper are the introduction of CCL (Section III), including a brief description of its semantics (Section IV), and the logic used to reason about CCL programs (Section V). The freedom with which schedules are used in the semantics of CCL is particularly new. We also present a detailed case study of the use of CCL with a simplified version of the above-mentioned capture-the-flag system, which we both specify (Section III-B) and verify (Section V-D). Ultimately we intend for CCL to be used not only for modeling and analysis of systems but also as a practical programming tool in its own right. Thus, we briefly describe a runtime CCL interpreter (Section VI), a prototype of which we have made available to the general community. We begin with a review of related work.

II. Related Work

The UNITY formalism [3] is used to specify and reason about concurrent reactive systems [14] and has been extended to real time systems [2]. Although UNITY was designed as a reasoning tool, an implementation of it has been built [15]. To the best of the author's knowledge, the application of UNITY-like formalisms to distributed control problems has not been well developed. Temporal logic has been used to specify and reason about concurrent and distributed systems as well as control systems [12].

Several efforts underway address the gulf between modeling, control and computation. Giotto [8] is a language for programming the interfaces between control-software modules. It makes explicit the timing, mode switching and communication aspects of an implementation that may have been hidden (or assumed) at the initial modeling and controller design stage. In CCL, one generally models and programs these aspects from the outset, although CCL could be used in conjunction with Giotto to schedule guarded command execution. Furthermore, CCL programs are required

to be robust to variations in low level scheduling (as in Definition 4.3). Charon [1] is modeling language that in essence formalizes state-charts (as in MATLAB). It provides a means of representing switching, parallel composition and refinement of continuous/discrete modes. In contrast, CCL adopts a logical/symbolic based description of both the system dynamics and the control code and is geared toward the specification of large scale distributed control systems. Reasoning in CCL is done with temporal logic and could be automated using either model checking or, more likely, theorem proving. The main difference between CCL and hybrid systems formalisms is that one can specify only discrete-time dynamical systems in CCL: Continuous dynamics must be approximated or discretized. We believe that this actually simplifies specifications while nevertheless maintaining the essence of the systems to be specified.

CCL originated with the practical desire to specify and program distributed robotic and control systems, such as the *Caltech Multi-vehicle Wireless Testbed* [4] or automated factories [10], in a principled yet natural manner. CCL has since grown into a convenient tool for specifying multi-agent control systems. CCL was first described in [11] where it is used to specify multi-robot communication algorithms whose *communication complexities* are subsequently analyzed.

III. CCL

In this section, we describe the Computation and Control Language (CCL) informally and present a detailed example to illustrate some aspects of the language. In later sections we fill in many of the details left out here.

A. Anatomy of a CCL Specification

A CCL specification, or *program*, P consists of two parts I and C. I is a predicate on states called the initial condition. C is a set of guarded commands of the form $g : r$ where g is a predicate on states and r is a *relation* on states. In a rule, primed variables (such as x_i') refer to the new state and unprimed variables to the old state. For example,

$$x < 0 : x' > y + 1$$

is a guarded command stating: If x is less than zero, then set the new value of x to be greater than the current value of y plus 1. If x is not less than zero, do nothing.

Programs are composed in a straightforward manner: If $P_1 = (I_1, C_1)$ and $P_2 = (I_2, C_2)$ then $P_1 \circ P_2 = (I_1 \wedge I_2, C_1 \cup C_2)$.

Programs in CCL are thus similar in appearance to UNITY specifications [3], except that we allow rules to be relations instead of assignments. However, due to our desire to model real-time, controlled systems, the semantics (i.e. interpretation) of CCL are somewhat different. To explain our choice of semantics for CCL, we first describe the semantics of UNITY.

> **UNITY Semantics:** At state s, nondeterministically choose a command $g : r$ from C. If g is $true$ in state s, then choose s' such that $s\ r\ s'$ is $true$. Repeat with the new state s'. Every command must be chosen infinitely often in any execution.

The benefit of this interpretation is that arbitrary interleaving of commands (supposedly each assigned to a different processor) can occur — modeling the possibility for each command to execute at a possible different rate. If one can reason that any interleaving leads to a correct behavior, then one has a good specification of a parallel system. However, in CCL we often combine computation *and* control commands (performed by control processors or robots), such as

$$g(x_i) \ : \ u_i' = h(\mathbf{x}), \tag{1}$$

with commands that model the environment as in

$$true \ : \ ||\mathbf{x}' - f(\mathbf{x}, \mathbf{u})|| < \varepsilon. \tag{2}$$

In the UNITY semantics, the latter command may be chosen one billion times before the former is chosen at all. Of course, one could change the guards and add an auxiliary synchronizing state so that the intended behavior occurs. This is essentially what we enforce with the CCL semantics.

> **CCL EPOCH Semantics:** Commands are chosen non-deterministically from C, but each command must be chosen once before any command is chosen again.

Thus, the execution of a system is divided into *epochs* during which each command is executed exactly once. This is an attempt[1] to capture the small-time interleaving that may occur between processors that are essentially synchronized. It is important to note that the set of behaviors of a specification under the CCL semantics is a subset of the behaviors of the same program under the UNITY semantics. Thus, any statement (in temporal logic) that is true about UNITY behaviors is true about the CCL behaviors. CCL behaviors may satisfy more properties, however, as discussed in Section V and demonstrated in Section V-D.

Remark on Time: The intention is that epochs occur at some fixed frequency, although this is not modeled explicitly. Thus, equation (2) would represent a periodic sampled version of some continuous-time system.

B. The RoboFlag Drill

In this section we consider a game called *RoboFlag*, which can be thought of as "capture the flag" for robots [5]. Two teams of robots, say *red* and *blue*, each have a defensive zone that they must protect (it contains the team's flag). If a red robot enters the blue team's defensive zone without being

[1]The EPCOH semantics are just one of several interpretations of CCL that we are exploring [9]. Other possibilities are that each command executes at approximately the same frequency; or each command executes equally many times in any interval.

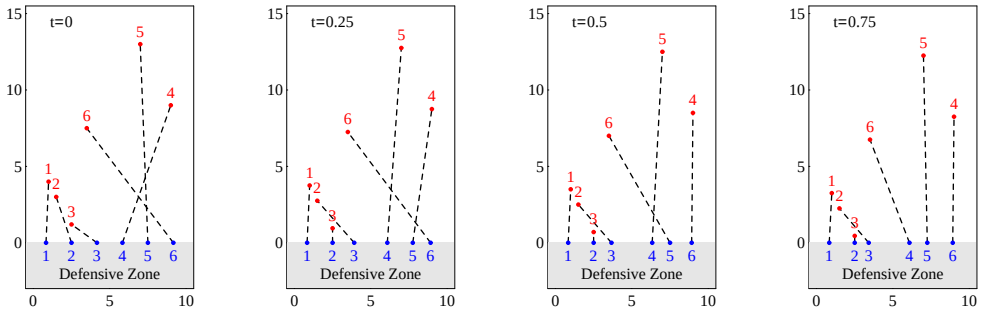

Fig. 1. The first four epochs of an execution of $P_{rf}(6)$. Dots along the x-axis represent blue defending robots. Other dots represent red attacking robots. Dashed lines represent the current assignment.

tagged by a blue robot, it captures the blue flag and earns a point. If a red robot is tagged by a blue robot in the vicinity of the blue defensive zone, it is disabled. These rules hold when the roles are reversed as well. We do not propose to address 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 1.

We first specify in Program $P_{red}(i)$ 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 $y_i \leq \delta$). 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	$x_i \in [min, max] \wedge y_i > \delta$		
Commands	$y_i - \delta > 0$	:	$y_i' = y_i - \delta$

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 (in an attempt to intercept it), as long as taking such an action does not lead to a collision.

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

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

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

To update 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 $z_i < z_j$ and $\alpha(i) > \alpha(j)$, 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. The following definition captures the latter situation.

$$r(i,j) \triangleq \begin{cases} 1 \text{ if } y_{\alpha(j)} < |z_i - x_{\alpha(j)}| - \delta \\ 0 \text{ otherwise.} \end{cases} \quad (3)$$

Finally, 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 intercepted or leaves it the same and decreases the number of conflicts.

$$\begin{aligned} switch(i,j) \quad \triangleq \quad & r(i,j) + r(j,i) < r(i,i) + r(j,j) \\ \vee \quad & (r(i,j) + r(j,i) = r(i,i) + r(j,j) \\ & \wedge x_{\alpha(i)} > x_{\alpha(j)}. \end{aligned}$$

The protocol is then

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

and the full RoboFlag drill system is given by

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

IV. The Semantics of CCL

A. Basic Definitions

We begin with a set V of *variable symbols* and a set Val of values that the variables may take. The set Val contains natural numbers, real numbers, sets, etc. We suppose that the type of a variable does not change, even though its value might. The definitions of *state*, *state function* and *action* in this subsection and the definition of *behavior* in the next are taken from [12].

A **state** s is a function from V to Val. The value of a variable $v \in V$ with respect to a state s is denoted $s[\![v]\!]$. A **state function** f is an expression over symbols in V and constant symbols. The set of all states is denoted S. The

meaning of a state function f, denoted $[\![f]\!]$, is a function from states into values and is defined by

$$s[\![f]\!] \triangleq f\,(\forall v \;:\; s[\![v]\!] \;/\; v),$$

that is, the value obtained by replacing all (free occurrences of) variables in f by their values under the state s. A **predicate** is simply a boolean valued state function.

We denote by V' the set $\{v' \;:\; v \in V\}$, that is, the set of all primed variables symbols from V. We assume $V \cap V' = \emptyset$. An **action** is a boolean valued expression over variables in V, primed variables in V' and constant symbols. If f is a state function then f' denotes $f(\forall v \;:\; v' \;/\; v)$. If p is a predicate, then p is also an action (with no primed variables) and p' is an action with only primed variables. The **meaning** of an action a, denoted $[\![a]\!]$, is a function from $S \times S$ into values and is defined by

$$s[\![a]\!]t \triangleq a\,(\forall v \;:\; s[\![v]\!] \;/\; v,\; t[\![v]\!] \;/\; v'),$$

that is, the value obtained by replacing all (free occurrences of) unprimed variables in a by their values under the state s and replacing all (free occurrences of) primed variables in a by their values under t. For example,

$$s[\![x' > x + 2y - 1]\!]t \equiv t(x) > 2s(x) + s(y) - 1.$$

Remark We usually regard variables not appearing primed in an action as not changing. So, for example, $x' > x + 1$ refers to

$$x' > x + 1 \;\wedge\; \forall v \in V - \{x\} \;.\; v' = v.$$

A particular kind of action is the **guarded command**, consisting of a predicate g (the guard) and an action r (the rule), usually written $g : r$. The **meaning** of $g : r$ is a function from $S \times S$ into values and is defined by

$$s[\![g : r]\!]t \triangleq (s[\![g]\!] \wedge s[\![r]\!]t) \vee (\neg s[\![g]\!] \wedge s = t).$$

We give the name *skip* to the guarded command

$$\mathit{true} : \forall v.v' = v \tag{4}$$

Note that *skip* is equivalent to the guarded command $\mathit{false} : a$ for any action a. Finally, a **program** or **specification** is a pair $P = (I, C)$ where I is a predicate called the **initial condition** and C is a set of guarded commands. If $P_1 = (I_1, C_1)$ and $P_2 = (I_2, C_2)$ are programs, their **composition** is defined by

$$P_1 \circ P_2 = (I_1 \wedge I_2, C_1 \cup C_2).$$

B. Behaviors and Schedules

Definition 4.1: A **behavior** is a sequence $\sigma : \mathbb{N} \to S$ of states. We denote $\sigma(k)$ by σ_k. A **schedule** for a program $P = (I, C)$ is a function $\omega : \mathbb{N} \to C \cup \{\mathit{skip}\}$ that assigns to each step either a guarded command from C or the command *skip* in Equation (4). The behavior σ is called an ω-**behavior** for the program $P = (I, C)$ if

i) $\sigma_0[\![I]\!]$

ii) $\forall k \;.\; \sigma_k[\![\omega(k)]\!]\sigma_{k+1}$.

If $\omega(k) = \mathit{skip}$ then we call step k a **stutter step**.

Based on behaviors, we will provide two ideas of what it means for a behavior σ to satisfy a program P (other less restrictive semantics are defined in [9]). For each semantics (i.e. meaning) M we will define the M-meaning of P, denoted

$$[\![P]\!]_M : (\mathbb{N} \to S) \to \{\mathit{true}, \mathit{false}\}$$

to be a boolean valued function on behaviors. We use the usual prefix notation $\sigma[\![P]\!]_M$ to denote the (boolean) value that $[\![P]\!]_M$ assigns to σ.

The least restrictive schedule for a program is one that eventually applies each guarded command, interleaving them in any order, as in UNITY [3]. The fact that each guarded command is eventually applied is equivalent to saying that the system enjoys *weak fairness* [12].

Definition 4.2: (UNITY SEMANTICS) Given a program $P = (I, C)$ and a behavior σ, then $\sigma[\![P]\!]_{UNITY} = \mathit{true}$ if and only if there exists a schedule ω such that

i) σ is an ω-behavior for P

ii) for all $c \in C$, the set $\omega^{-1}(c)$ is infinite.

However, a reasonable assumption for scheduling a program is that the entire set of guarded command is fired over and over again. That is, that no command is fired again until all commands have been fired once. The set of steps during which each command is fired once is called an *epoch*. This notion is defined formally in the next definition.

Definition 4.3: (EPOCH SEMANTICS) Given a program $P = (I, C)$ and a behavior σ, then $\sigma[\![P]\!]_{EPOCH} = \mathit{true}$ if and only if there exists a schedule ω such that

i) σ is an ω-behavior for P

ii) there exists an increasing sequence of natural numbers $\{n_i\}_{i \in \mathbb{N}}$ with $n_0 = 0$ such that for all i, if $n_i \leq k < l < n_{i+1}$ then either

$$\omega(k) \neq \omega(l) \quad \text{or} \quad \omega(k) = \omega(l) = \mathit{skip}.$$

The subsequence $\langle \sigma_{n_i}, ..., \sigma_{n_{i+1}-1} \rangle$ is called the ith **epoch** of σ under ω.

V. Properties of Specifications

A. Temporal Logic

In this and the following sections we will need to reason about the effect of an action on the set of all states satisfying a predicate. For this, we use the *Hoare triple* notation (standard in Computer Science) defined in CCL as follows.

Definition 5.1: Let a be an action and let P and Q be predicates. Then the Hoare triple relating P to Q by a is defined as

$$\{P\}\; a\; \{Q\} \triangleq \forall s, t \;.\; s[\![P]\!] \wedge s[\![a]\!]t \Rightarrow t[\![Q]\!].$$

We reason about entire CCL programs using standard temporal logic [13], [12], which due to space constraints and the desire not to be redundant, we do not describe completely here. Briefly, *temporal logic formulas* are constructed from predicates, actions, basic connectives (such as $\vee$, $\wedge$, $\neg$ and $\Rightarrow$) and the special operators $\Box$ (always) and $\Diamond$ (eventually). Given a temporal logic formula F, we define $[\![F]\!]$ to be a function from behaviors to $\{true, false\}$ and say that σ satisfies F if $\sigma[\![F]\!]$. If p is a predicate, a an action, and F and G be arbitrary temporal logic formulas, then

i) $\sigma[\![p]\!] \triangleq \sigma_0[\![p]\!]$,
ii) $\sigma[\![a]\!] \triangleq \sigma_0[\![a]\!]\sigma_1$,
iii) $\sigma[\![\neg F]\!] \triangleq \neg\sigma[\![F]\!]$,
iv) $\sigma[\![F \wedge G]\!] \triangleq \sigma[\![F]\!] \wedge \sigma[\![G]\!]$,
v) $\sigma[\![\Box F]\!] \triangleq \forall n.\langle\sigma_n, \sigma_{n+1}, ...\rangle[\![F]\!]$.

The formula $\Diamond F$ is equal to $\neg\Box\neg F$. The following properties are useful in reasoning about programs. They are similar to those introduced in [3] and [14].

Definition 5.2: Let p and q be predicates. Then

i) p **co** $q \triangleq \Box(p \Rightarrow [(q' \vee skip) \wedge \Diamond q'])$
ii) $p \rightsquigarrow q \triangleq \Box(p \Rightarrow \Diamond q)$.

Thus, p **co** q (read "p constrains q") means that whenever p is true, then after the next time the state changes, q will be true. The second property, $p \rightsquigarrow q$ (read "p leads to q") means that whenever p is true, q will be true at some later time.

One of the most common properties we desire of a CCL program is that starting from any initial state allowed by I, some property F is eventually always true, that is $\Diamond\Box F$. Thus, if the system is disturbed from equilibrium (F), it will eventually return. We call this "strong stability" (as opposed to just stability which, in computer science, is $\Box F$). The rules for showing strong stability are summarized in the following lemma.

Lemma 5.1: Let F be a temporal logic formula, V be a state function over the natural numbers, including zero, and σ be any behavior. Then

i) For any $k \in \mathbb{N}$, if $\sigma[\![V = k \rightsquigarrow (G \vee V < k)]\!]$ then $\sigma[\![V = k \rightsquigarrow G]\!]$.
ii) For any k, if $\sigma[\![V = k \rightsquigarrow F]\!]$ and $\sigma[\![F$ **co** $F]\!]$ then $\sigma[\![\Diamond\Box F]\!]$.

We use temporal logic formulas to specify the behaviors we desire for the CCL programs we write. Thus, we would like to know when all behaviors allowed by a particular program also satisfy a given temporal logic formula. This is captured in the following definition in which M ranges over $UNITY$ and $EPOCH$.

Definition 5.3: Let P be a CCL program and F a temporal logic formula. Then P **models** F with respect to the semantics M, written $P \models_M F$, if and only if

$$\forall\sigma \,.\, \sigma[\![P]\!]_M \Rightarrow \sigma[\![F]\!].$$

If $P \models_{UNITY} F$ we just write $P \models F$. Note that $P \models F$ implies that $P \models_{EPOCH} F$, but not conversely.

B. Reasoning About UNITY

We wish to determine when $P \models F$ by examining the initial condition and the commands of P. We first have a lemma relating the commands in a program P with the **co** relation.

Lemma 5.2: Let $P = (I, C)$ be a program and p and q be predicates. If for all commands $c \in C$ we have $\{p\}\ c\ \{q\}$ then $P \models p$ **co** q.

That is, whenever the effect of any guarded command executed in a state for which p is true results in a state for which q is true, we may conclude that p **co** q.

The formula p **co** p implies that once p becomes true, it stays true. This is reflected in the following lemma:

Lemma 5.3: Let $P = (I, C)$ be a program and p be a predicate. Then if $I \Rightarrow p$ and $P \models p$ **co** p, then $P \models \Box p$.

The next lemma is a rule for showing progress.

Lemma 5.4: Let $P = (I, C)$ be a program and p and q be predicates. If $\{p\}\ c\ \{p \vee q\}$ for all $c \in C$ and if there exists a $d \in C$ such that $\{p\}\ d\ \{q\}$, then $P \models p \rightsquigarrow q$.

C. Reasoning About EPOCH

Determining when $P \models_{EPOCH} F$ even if $P \not\models F$ (under the *UNITY* semantics), requires examining behaviors on an epoch by epoch basis, as the inference rule in this section implies. We use the notation $\Pi(C)$ for the set of all permutations (bijections) $\pi : \{1, ..., |C|\} \rightarrow C$.

Lemma 5.5: Let $P = (I, C)$ be a program with $C = \{c_1, ..., c_n\}$ and let p be a predicate. If there exists a predicate q such that

i) $I \Rightarrow q$;
ii) $q \Rightarrow p$;
iii) for all $\pi \in \Pi(C)$ it is the case that

$$\{q\}\ c_{\pi(1)}\ \{p \vee q\}\ c_{\pi(2)}\ ...\ c_{\pi(n-1)}\ \{p \vee q\}\ c_{\pi(n)}\ \{q\};$$

then $P \models_{EPOCH} \Box p$.

D. Properties of the RoboFlag System

We now verify several properties of $P_{rf}(n)$, defined in Section III-B. In particular, we show that (1) no two blue robots collide; (2) under certain assumptions, the assignment protocol stabilizes (eventually no switches ever occur again); and (3) under certain restrictions on the initial positions of the robots, the protocol stabilizes *before* the red robots reach the defensive zone. Due to space considerations, we only sketch the proofs. First, we show that no two blue robots collide.

Theorem 5.1: $P_{rf}(n) \models \Box z_i < z_{i+1}$.

Proof (Sketch): We first use Lemma 5.2 to show that $z_i < z_{i+1}$ **co** $z_i < z_{i+1}$. Given i, the only commands that change z_i or z_{i+1} are the second command in $P_{blue}(i)$ and the first in $P_{blue}(i+1)$. In the first case $\{z_i < z_{i+1}\}\ c\ \{z_i < z_{i+1}\}$ is equivalent to

$$z_i < z_{i+1} \wedge \big(z_i < x_{\alpha(i)} \wedge z_i < z_{i+1} - \delta\ :\ z_i' = z_i + \delta\big) \Rightarrow z_i' < z_{i+1}'$$

which is easily shown to be true using the definition of guarded command. A similar argument shows that the first command of $P_{blue}(i+1)$ also preserves $z_i < z_{i+1}$. With the trivial fact that $I_{rf}(n) \Rightarrow z_i < z_{i+1}$, we have the desired result using Lemma 5.3. ∎

Next, we show that the assignment protocol *self stabilizes*. That is, we show that eventually no more assignment switches will be made. For convenience, define a predicate Q (for "quiescent") by

$$Q \triangleq \forall i.\neg switch(i, i+1). \tag{5}$$

To prove the result, we make use of a *Lyapunov* style argument (as in Lemma 5.1), showing that a non-negative function decreases with each switch. First define

$$\rho \triangleq \sum_{i=1}^{n} r(i,i)$$

to be the total number of blue robots that are too far away to reach their assigned red robots, assuming all the robots move at the same rate. And define

$$\beta \triangleq \sum_{i=1}^{n}\sum_{j=i+1}^{n} \gamma(i,j), \text{ where } \gamma(i,j) \triangleq \begin{cases} 1 \text{ if } x_{\alpha(i)} > x_{\alpha(j)} \\ 0 \text{ otherwise,} \end{cases}$$

to be the total number of conflicts in the current assignment. Noting that $\rho \in \{0, ..., n\}$ and $\beta \in \{0, ..., \binom{n}{2}\}$, we define V by

$$V \triangleq \left[\binom{n}{2} + 1\right]\rho + \beta.$$

Note that $V \geq 0$. Our goal is to show that V does not increase. This is true, however, only under certain conditions, a full analysis of which are beyond the scope of this paper. Instead, we will assume that the system is "safe" in the simple (and very conservative!) sense defined by

$$Safe \triangleq \Box\left(\forall i\ .\ r(i,i) = 0 \wedge z_{i+1} - z_i > 2\delta\right).$$

Although $Safe$ is true only for certain behaviors allowed by P_{rf}, the protocol stabilizes for many others that we do not describe. Essentially (with $Safe$) we are avoiding the situation where two blue robots almost collide, and therefore make no progress toward their assignments; and the situation where a red robot alternates between being far enough for its defender to intercept and too close to intercept in a single epoch. In any case, a simple stabilization theorem is as follows.

Theorem 5.2: $P_{rf}(n) \models Safe \Rightarrow \Diamond\Box Q$.

Proof (Sketch): We first show that for any $k \in \mathbb{N}$,

$$P_{rf}(n) \models Safe \Rightarrow (V = k \rightsquigarrow (V < k \vee Q)) \tag{6}$$

using Lemma 5.4. Since the formula $Safe$ holds, none of the *motion* commands in $P_{red}(n)$ or $P_{blue}(n)$ will decrease V. Thus, for all the motion commands c, we have $\{V = k\}\ c\ \{V = k\}$. Also, given that $V = k$ and that Q is false (otherwise we are done), there must be a command c in $P_{proto}(n)$ whose guard is true. The effect of this clause is to decrease V since it clearly decreases β. Thus, for this clause, $\{V = k\}\ c\ \{V < k\}$ so that (6) holds by Lemma 5.4. The result then follows using (6) and Lemmas 5.2 and 5.1. ∎

Last, we show that if the red robots start "far enough" away, then the assignments stabilize before any red robot reaches the defensive zone. We first define what "far enough" means:

$$Far \triangleq \forall i.y_i > \delta\kappa_n + \delta \wedge x_i + 2\delta\kappa_n \leq x_{i+1} \tag{7}$$

where κ_n is a the maximum value of V:

$$\kappa_n \triangleq (n+1)\binom{n}{2} + n.$$

Note that, since Far is a predicate, it applies only to the first state of any behavior for which the specification applies. Thus, it will be a restriction on the initial condition of $P_{rf}(n)$. Also, note that the protocol can only stabilize before the red robots get to the defensive zone under the assumption that the protocol commands and the red robot commands are executed equally often. Thus we will need the *EPOCH* semantics for the following theorem.

Theorem 5.3: $P_{rf}(n) \models_{EPOCH} Far \wedge Safe \Rightarrow \Box\left(\neg Q \Rightarrow \forall i.y_i > \delta\right)$.

Proof (Sketch): With the fact that $\neg Q \Rightarrow V > 0$, it suffices to show that

$$P_{rf}(n) \models_{EPOCH} Far \wedge Safe \Rightarrow \Box(\forall i\ .\ y_i > \delta V).$$

We will use Lemma 5.5 with q defined to be the predicate $y_i - \delta > \delta V$. Because $\neg Q$, some switch will occur to decrease V. Because $Safe$ and Far, no switch will occur to increase V. Now let $i \in \mathbb{N}$. All commands excpet the one in $P_{red}(i)$ leave y_i the same and all commands except the active switch command(s) in $P_{proto}(n)$ will decrease V by at least one. We need only consider whether the red clause occurs first, or after one of the switch commands. In both cases we have that q remains true. At the end of the epoch, y_i will have decreased by δ and V will have decreased by at least one, thus part (*iii*) of Lemma 5.5 is satisfied. The other preconditions of the lemma are simple. ∎

VI. CCL Software Tools

In addition to providing a tool for modeling systems and a logic for reasoning about such models, we are developing a programming language version of CCL for the implementation of controllers in settings where control and communication depend on each other. To this end, we have built a prototype CCL interpreter called CCLi. CCLi input consists of basic types (boolean, integer, floating point, string, list, lambda expression and records), a rich set of possible expressions on these types, and *programs*. A program is essentially a specification as defined in Section IV except that rules are sets of assignments (as opposed to arbitrary relations as in Section IV). Also, programs may be composed with *hidden* and *shared* variables as in

$$P_1 \circ P_2 \; sharing \; x, y$$

which means that any use of variables named x or y in P_1 and P_2 refer to the same object, while all other variable references are local to P_1 or P_2 as the case may be.

CCLi includes an interface to shared libraries (written in C++ for example) so that it can easily be interfaced to libraries in other languages, simulation software or a controls testbed. We are presently experimenting with CCLi program development with the Caltech Multi-Vehicle Wireless Testbed [4]. More information about CCLi, including many examples such as the encoding of the RoboFlag Drill from this paper, can be found at http://www.cs.caltech.edu/~klavins/ccl/.

VII. Conclusion

We have introduced a distributed systems based formalism, CCL, for modeling control systems wherein computation takes a primary role. We demonstrated CCL by specifying and verifying the RoboFlag drill controller and its self-stabilizing protocol. This example is typical of the sorts of systems that we wish to specify and build, involving both control (albeit extremely simple in this example) and distributed computation.

The verification of high level algorithms such as the one presented in this paper, however, represents only the first step on the design path. We are presently exploring methods for refining CCL specifications into executable code by applying the formalism to increasingly rich examples from, for example, our multi-vehicle testbed [4] where we plan to have real vehicles running verified CCL programs.

The CCL formalism also presents certain difficult challenges. Although it is easy to represent the idea of strong stability ($\Diamond\Box F$) from which standard notions of stability in control (e.g. asymptotic stability) can be defined, other notions crucial to control theory are not so straightforward. For example, the notion of robustness to model uncertainty is in particular not expressible in CCL (or any other related formalism), because logical and interleaved-execution models do not admit naturally metrizable space. Recent results in concurrency theory [6], where metrics are defined on discrete probabilistic transition systems, may be applicable to this problem, however.

Acknowledgments

Many of the ideas in this paper grew out of discussions with Jason Hickey, Richard M. Murray, Raff D'Andrea and Reza Olfati-Sabor. Natarajan Shankar made several suggestions regarding self-stabilizing protocols. This research is supported in part by the AFOSR, grant number F49620-01-1-0361.

VIII. References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifi cations of hybrid systems in CHARON. In *Hybrid Systems: Computation and Control*, LNCS 1790, pages 6–19, Pittsburgh, PA, 2000. Springer-Verlag.

[2] A. Carruth. Real-time UNITY. Technical Report TR-94-10, University of Texas at Austin, 1994.

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

[4] L. Cremean, B. Dunbar, D. van Gogh, J. Hickey, E. Klavins, J. Meltzer, and R. M. Murray. The caltech multi-vehicle wireless testbed. In *41st IEEE Conference on Decision and Control*, Las Vegas, NV, December 2002.

[5] 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.

[6] J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangadon. The metric analogue of weak bisimulation for probabilistic processes. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 413–422, 2002.

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

[8] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software*, LNCS 2211, pages 166–184. Springer-Verlag, 2001.

[9] E. Klavins. The computation and control language (CCL). In Progress.

[10] E. Klavins. Automatic compilation of concurrent hybrid factories from product assembly specifi cations. In *Hybrid Systems: Computation and Control*, LNCS 1790, pages 174–187, Pittsburgh, PA, 2000. Springer-Verlag.

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

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

[13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specifi cation*. Springer-Verlag, 1992.

[14] J. Misra. A logic for concurrent programming: Safety and Progress. *Journal of Computing and Software Engineering*, 3(2):239–300, 1995.

[15] R. T. Udink and J. N. Kok. Impunity: UNITY with procedures and local variables. In *Mathematics of Program Construction*, pages 452–472, 1995.