A Formal Model of a Multi-Robot Control and Communication Task

Eric Klavins[1]

Computer Science Department
California Institute of Technology, Pasadena, CA
klavins@caltech.edu

Abstract

We introduce the Computation and Control Language (CCL), a *guarded-command* language for expressing systems wherein control and computation are intertwined. A CCL program consists of a set of guarded commands that may update continuous or discrete variables and that can be reasoned about using a simple *temporal logic*. In this paper, a detailed case study of a robot capture-the-flag system, called the "RoboFlag Drill", is encoded in CCL and certain 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.

1 Introduction

We are interested in designing large scale decentralized systems with multiple computational and controlled elements such as multi-vehicle systems 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 3.2) involves a group of robots that must defend their flag against a group of opponent robots in a capture-the-flag like game. 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 [6] under certain circumstances (explored in Section 6) 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 this system in a formalism called the *Computation and Control Language* (CCL). CCL is inspired by the UNITY formalism for parallel programming [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 the 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 to be used for modeling, specification and analysis.

The main contributions of the paper are the introduction of CCL (Section 3) including a formal description of its semantics (Section 4) and the logic used to reason about CCL programs (Section 5). The *static* operator (Definition 5.4) that deals with arbitrary schedules is, in particular, 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 3.2) and verify (Section 6). Ultimately we intend for CCL to be used to not only for modeling and analysis of systems but also as a programming tool in its own right. Thus we also describe a runtime CCL interpreter (Section 7). We begin with a review of related work.

2 Related Work

The UNITY formalism was introduced in [3] and is used to specify and reason about concurrent reactive systems [13] and has been extended to real time systems [2]. Although UNITY was designed as a reasoning tool, an

[1]This research is supported in part by AFOSR grant number F49620-01-1-0361.

implementation of it has been built [17]. To the best of the author's knowledge, the *static* operator (Definition 5.4) is defined in this paper for the first time.

There are several efforts underway that address the gulf between modeling, control and computation. Giotto [7] 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. Furthermore, CCL programs are required to be (in the sense defined in Section 5) robust to low level scheduling. Charon [1] is modeling language that in essence formalizes state-charts, a tool provided by 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. Furthermore, one can specify only discrete-time dynamical systems in CCL and must approximate continuous dynamics with discretizations of continuous systems. This is similar to, for example, the *tick-rules* in real-time Maude [14].

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 [8, 9], in a principled yet natural manner. It was loosely inspired by *RHexLib*, a C++ library [11] for programming the *RHex Robot* [16]. CCL has since grown into a convenient tool for specifying multi-agent control systems. CCL was first described in [10] where it is used to specify multi-robot communication algorithms whose *communication complexities* are subsequently analyzed.

3 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.

3.1 Anatomy of a CCL Specification

A CCL specification Π consists of two parts I and C. I is a predicate on states called the initial condition. C is a set of guarded commands, or *clauses*, of the form $g : r$ where g is a predicate on states and r is a *relation* on states. Specifications 1, 2 and 3, described in the next subsection, are examples. Note that in the rules, primed variables (such as x_i') refer to the new state and unprimed variables to the old state. Specifications are composed in a straightforward manner (as in equations (4) and (6) defined in the next subsection): If $\Pi_1 = (I_1, C_1)$ and $\Pi_2 = (I_2, C_2)$ then $\Pi_1 \circ \Pi_2 = (I_1 \cap I_2, C_1 \cup C_2)$.

Specifications 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 of CCL are somewhat different. To explain our choice of semantics, we first describe the semantics of UNITY.

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

The benefit of this interpretation is that arbitrary interleavings of clauses (supposedly each assigned to a different processor) can occur. 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 usually combine computation *and* control clauses (performed by control processors or robots), such as

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

with clauses that model the environment as in

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

In the UNITY semantics, the latter clause 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 Semantics: At state $s' = s_0$, choose an ordering $g_0 : r_0$, $g_1 : r_1$, ..., $r_{|C|-1} : g_{|C|-1}$ of the clauses. Obtain new states s_1, s_2, ..., $s_{|C|-1} = s'$ such that if $g_k(s_k)$ then $r(s_k, s_{k+1})$.

Thus, the execution of a system is divided into *epochs* during which each clause is executed exactly once. This is an attempt to capture the small-time interleaving that may occur between processors executing at essentially the same rate. 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 5.

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.

3.2 Example: The RoboFlag Drill

In this section we consider a game called *RoboFlag* that is similar to "capture the flag", only for robots [5]. Two

Specification 1 Red Robot Dynamics: $\Pi_{red}(i)$

Initial:

$x_i \in [min, max] \wedge y_i > \max$

Clauses:

$y_i - \delta > 0 \ : \ y_i' = y_i - \delta$

Specification 2 Blue Robot Control: $\Pi_{blue}(i)$

Initial:

$z_i \in [min, max] \ \wedge \ z_i < z_{i+1}$

Clauses:

$z_i < x_{\alpha(i)} \wedge z_i < z_{i+1} - 2\delta \ : \ z_i' = z_i + \delta$

$z_i > x_{\alpha(i)} \wedge z_i > z_{i-1} + 2\delta \ : \ z_i' = z_i - \delta$

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 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 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 1.

We first specify $\Pi_{red}(i)$ in Specification 1, the very simplified dynamics of red robot i. It simply moves toward the defensive zone with constant vertical velocity of $-\delta$ m/s and no horizontal movement. When it reaches the defensive zone, it simply 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. The specification is parameterized by $i \in \mathbb{N}$ and thus describes a family of specifications.

The reactive control law, $\Pi_{blue}(i)$ in Specification 2, 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 applies the law

$$\dot{x}_i = -\text{sgn}(x_i - z_{\alpha(i)}) \tag{3}$$

for δ seconds, as long as taking such an action does not lead to a collision. The dynamics of the entire drill system are defined by

$$\Pi_{drill}(n) \triangleq \Pi_{red}(1) \circ ... \circ \Pi_{red}(n) \circ \Pi_{blue}(1) \circ ... \circ \Pi_{blue}(n). \tag{4}$$

The success of the control law as it stands depends very much on the initial positions of the red robots and the assignment α. Thus, we have devised a simple protocol,

Specification 3 Assignment Protocol: $\Pi_{proto}(n)$

Initial:

α is a bijection from $\{1, ..., n\}$ to $\{1, ..., n\}$.

Clauses:

$switch_{1,2} \ : \ (\alpha(1)', \alpha(2)') = (\alpha(2), \alpha(1))$

...

$switch_{n-1,n} \ : \ (\alpha(n-1)', \alpha(n)') = (\alpha(n), \alpha(n-1))$

$\Pi_{proto}(n)$ in Specification 3, for updating α. Each robot negotiates with its left and right neighbors to determine whether it should trade assignments with one of them. We first make the following definition

$$r_{i,j} \triangleq \begin{cases} 1 \text{ if } y_{\alpha(j)} < |z_i - x_{\alpha(j)}| - \delta \\ 0 \text{ otherwise.} \end{cases} \tag{5}$$

Thus, $r_{i,j} = 1$ when it is *not possible* for blue robot i to reach red robot $\alpha(j)$ before it reaches the defensive zone (assuming no blue-blue robot collisions). The protocol also uses the notion of a *conflict*: Blue robots i and j are in conflict if $z_i < z_j$ but $x_{\alpha(i)} > x_{\alpha(j)}$. We define the predicate $switch(i, j)$ by

$$\begin{aligned} switch_{i,j} \ &\triangleq \ r_{i,j} + r_{j,i} < r_{i,i} + r_{j,j} \\ &\vee \ \left(r_{i,j} + r_{j,i} = r_{i,i} + r_{j,j} \ \wedge \ x_{\alpha(i)} > x_{\alpha(j)}\right). \end{aligned}$$

Thus, $switch_{i,j}$ is true exactly when 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 full system is given by

$$\Pi_{rf}(n) \triangleq \Pi_{drill}(n) \circ \Pi_{proto}(n). \tag{6}$$

The protocol described in $\Pi_{proto}(n)$ is an example of a *self-stabilizing* protocol [6] in that it settles into a mode where no assignment trades are made after an initial transient period. 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. The conditions under which this is possible are examined in Section 6.

4 The Semantics of CCL

We define a **state** to be an assignment of values to variables. For example,

$$s = \langle x := 1; \ p := \mathit{true}; \ t := 0.99 \rangle$$

is a state over the variable set $V = \{x, p, t\}$. Each variable $v \in V$ has a type $type(v)$ and we assume the values assigned by states are always correctly typed. The set of all states over a variable set V is denoted S_V, or just S when there is no danger of confusion.

A **predicate** on states is a set $P \subseteq S_V$ of states. For example,

$$g = \{s \mid s(x) > 0\}$$

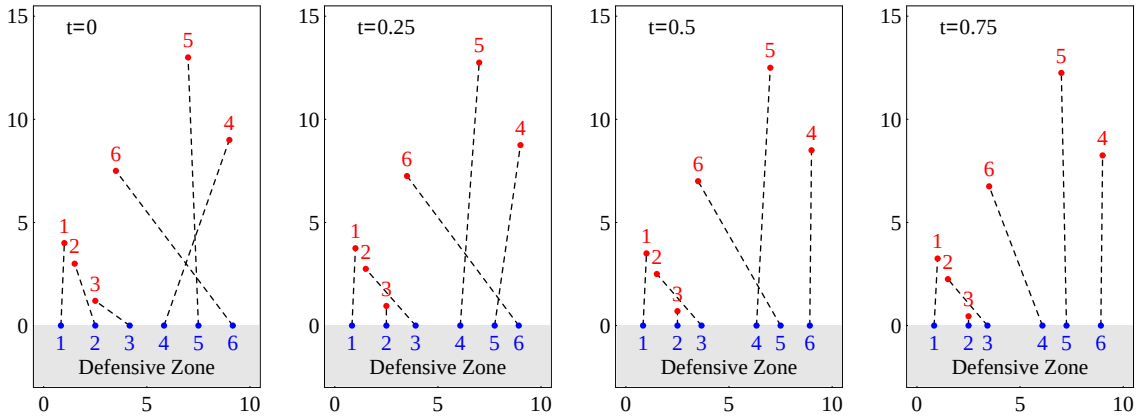

Fig. 1: The first four epochs of an execution of $\Pi_{rf}(6)$. Dots along the x-axis represent blue defending robots. Other dots represent red attacking robots. Dashed lines represent the current assignment. The trading of assignments of blue robots 2 and 3 reduces $r_{2,\alpha(2)} + r_{3,\alpha(3)}$. Other trades reduce the number of conflicts.

denotes the set of states that assign the variable x to a value greater than zero. We usually denote predicates in shorthand, so that the predicate g above is written

$$x > 0.$$

A **rule** on states is a relation $r \subseteq S_V \times S_V$ between states. For example

$$r = \{(s_1, s_2) \mid s_2(x) > s_1(x) \ \wedge \ s_2(t) = s_1(t) + \delta\}$$

is one such rule. As with predicates, we usually denote rules in shorthand as well so that the rule r above is written

$$x' > x \ \wedge \ t' = t + \delta.$$

The rule r does not refer to the variable p. The intention is that if a variable does not appear primed, it should stay the same (in this case $p' = p$) under application of the rule. We tacitly make this assumption in this paper.

The basic unit of a CCL specification is the *clause*. Formally, a **clause** is a pair

$$c = g : r$$

where g is a predicate called the **guard** and r is a rule. Given a clause c, we denote the guard and rule of c by $g(c)$ and $r(c)$ respectively. A clause defines a relation on states. For example, define the clause c by

$$x > 0 \ : \ x' \in [\frac{x}{2}, x] \tag{7}$$

where, as in our examples so far, $V = \{x, p, t\}$. Then c defines the relation

$$\Big(\Big(x > 0 \wedge x' \in [\frac{x}{2}, x]\Big) \vee (x \leq 0 \wedge x' = x)\Big)$$

$$\wedge\, p' = p \ \wedge \ t' = t.$$

For an arbitrary clause we have:

Definition 4.1 *If s and s' are states and $g : r$ is a clause, we say that s'* **simply follows** *s, denoted $s \xrightarrow{g:r} s'$, if either $g(s) \wedge r(s, s')$ or $\neg g(s) \wedge s' = s$.*

We now define specifications on states and their operational semantics as dynamical systems through state space.

Definition 4.2 *A CCL* **specification** *Π over a set of variables V is a pair*

$$\Pi = (I, C)$$

where I is a predicate over V and C is a set of clauses over V.

Definition 4.3 *Let $\Pi_1 = (I_1, C_1)$ and $\Pi_2 = (I_2, C_2)$ be specifications. Their* **composition** *is*

$$\Pi_1 \circ \Pi_2 = (I_1 \cap I_2, C_1 \cup C_2).$$

Definition 4.4 *A* **schedule** *for a specification $\Pi = (I, C)$ is a function $\omega : \{0, ..., |C| - 1\} \to C$.*

Definition 4.5 *Let Π be a specification and $s, s' \in S$. We say that s'* **follows** *s, denoted $s \xrightarrow{\Pi} s'$, if there exists a schedule ω and states s_0, ..., $s_{|C|-1}$ such that*

1. *$s = s_0$ and $s_{|C|-1} = s'$ and*
2. *for all k, $s_k \xrightarrow{\omega(k)} s_{k+1}$.*

Definition 4.6 *A* **behavior** *of a specification $\Pi = (I, C)$ is a sequence of states $\sigma : \mathbb{N} \to S$ such that*

1. *$\sigma(0) \in I$*

2. $\sigma(k) \xrightarrow{\Pi} \sigma(k+1)$.

The set of all behaviors of Π *is denoted* $\mathcal{E}(\Pi)$.

5 Properties of Specifications

Following [15], we will treat properties of specifications as sets of specifications. Thus, a specification Π has property P if $\Pi \in P$. We define standard notions of safety (*always* and *co*) and progress (*leadsto*) as well as the notion that a property is *static*, i.e. that it is true at all points *during* the execution of an epoch.

Definition 5.1 *Let A be a predicate. Then A is* **always** *true, written* $\Pi \in$ **always** A*, if and only if for all* $\sigma \in \mathcal{E}(\Pi)$ *and all* $k \in \mathbb{N}$*, we have that* $\sigma(k) \in A$.

Definition 5.2 *Let A and B be predicates. Then A* **constrains** *B, written* $\Pi \in A$ **co** B *if and only if*

$$\forall\, s_1, s_2 \,.\, (s_1 \in A \;\wedge\; s_1 \xrightarrow{\Pi} s_2) \Rightarrow s_2 \in B.$$

Definition 5.3 *Let A and B be predicates. Then A* **leads to** *B, written* $\Pi \in A \mapsto B$*, if and only if for all* $\sigma \in \mathcal{E}(\Pi)$ *and* $n \in \mathbb{N}$ *we have*

$$\sigma(n) \in A \Rightarrow \exists m \geq n \,.\, \sigma(m) \in \mathbb{N}.$$

Definition 5.4 *Let A be a property and* $\Pi = (I, C)$ *be a schedule. Then A is* **static***, written* $\Pi \in$ **static** A*, if and only if for all schedules* ω*, if*

$$s_0 \xrightarrow{\omega(0)} \ldots \xrightarrow{\omega(|C|-1)} s_{|C|-1}$$

and $s_0 \in A$ *then* $s_k \in A$ *for all* $k \in \{0, ..., |C|-1\}$.

These basic properties (except for *static*) obey all the inference rules that the corresponding UNITY properties do [3] (although the proofs are somewhat different). We illustrate a few (that we will encounter in Section 6) in the following proposition.

Proposition 5.1 *For any specification* $\Pi = (I, C)$ *and predicates A, B, A_1, A_2, B_1, B_2 and C,*

a. **static** $A \;\Rightarrow\; A$ **co** A

b. $I \rightarrow A \wedge$ **static** $A \;\Rightarrow\;$ **always** A

c. A_1 **co** B_1 *and* A_2 **co** $B_2 \Rightarrow A_1 \cap A_2$ **co** $B_1 \cap B_2$

d. A **co** $B \Rightarrow A \mapsto B$

e. $A \mapsto C$ *and* $B \mapsto C \Rightarrow (A \cup B) \mapsto C$.

We also have a variant of Lyapunov stability for specifications. The following proposition is similar to the LATTICE rule in [12].

Proposition 5.2 *If* $U : S \rightarrow \mathbb{N}$ *and*

$$\Pi \in \neg A \wedge U = k \textbf{ co } A \vee U < k$$

then $\Pi \in \neg A \wedge U = k \;\mapsto\; A$.

In the next section we use these propositions to help us show various safety, progress and stability properties of the RoboFlag drill.

6 Analysis of the RoboFlag Drill

We now verify several properties of the RoboFlag specification $\Pi_{rf}(n)$ defined by equation (6) in Section 3. Due to the space limitations of this paper, we do not include detailed proofs for these results, which usually require the analysis of many cases arising from the possible interleavings of the clauses. Instead, we endeavor to sketch the proofs of the properties below which are, after all, quite straightforward. We assume a fixed number n of blue(red) robots.

We first express the obvious fact that the system is safe: no robots will collide.

Theorem 6.1 *For all i, Π_{rf} satisfies the property* **always** $z_i < z_i + 1$.

Proof: Certainly $I \Rightarrow z_i < z_i + 1$. The only clauses that change z_j for any j are in Π_{blue}. These have guards to prevent their application if z_i and z_{i+1} are too close. Thus, Π_{rf} satisfies **static** $z_i < z_i + 1$. The result follows from Proposition 5.1(a). ∎

Next we show that the protocol is self stabilizing. For convenience, we need two auxiliary definitions. The *Active* property expresses the fact that the protocol Π_{proto} has not determined a stable assignment of blue to red robots:

$$Active \triangleq \exists i \in \{1, ..., n-1\} \,.\, switch_{i,i+1}. \qquad (8)$$

The $Safe(m)$ property, parameterized by the natural number m, states that no red robots will reach the defensive zone and no blue robots will collide in the next m epochs:

$$Safe(m) \triangleq \forall i \in \{1, ..., n\} \,.\, y_i > \delta \;\wedge\; z_i + 2\delta m < z_{i+1}. \qquad (9)$$

We can show that the protocol is quiescent under certain circumstances. Ideally, this means that $\neg Active$ **co** $\neg Active$. However, $\neg Active$ may not be

stable if some defender blocks another from moving toward its assigned red attacker (recall the guards in Specification 2). Such a situation may result in $r_{i,i}$ increasing for the blocked robot. Thus, we must show the weaker result that P is stable as long as the blue robots are sufficiently far apart.

Lemma 6.1 *If $m > 1$ then Π_{rf} satisfies the property Safe(m)* **co** *Safe$(m-1)$.*

Proof Sketch: Consider, for each i, the clauses affecting y_i and z_i. In particular, the clauses for y_i decrease y_i by δ and the clauses for z_i decrease the distance between z_i and z_{i+1} by 0, δ or 2δ. ■

Lemma 6.2 *If $m > 1$ then Π_{rf} satisfies the property* **static** *$[Safe(m) \vee Safe(m-1)] \wedge \neg Active$.*

Proof Sketch: For each i, consider all interleavings of the clauses affecting variables subscripted by i or $i+1$. In each case, all red robots move down and all blue robots move toward their assignments. Thus, at all steps either *Safe*(m) or *Safe*$(m-1)$. Furthermore, because $\neg Active$ and because $switch_{i,i+1}$ does not change as long as *Safe*(m) for some $m > 2$, $switch_{i,i+1}$ remains false for all i. ■

Theorem 6.2 *If $m > 1$ then Π_{rf} satisfies the property Safe$(m) \wedge \neg Active$* **co** *Safe$(m-1) \wedge \neg Active$.*

Proof: By Lemma 6.1, Lemma 6.2 and Proposition 5.1(a) and (c). ■

We next show that if the the protocol begins when the red robots are "far enough" away then it will self-stabilize before they arrive. To do this, we will show that a certain function of **x**, **y** and **z** decreases every time an assignment is updated by one of the rules in $\Pi_{proto}(n)$. First define

$$\rho \triangleq \sum_{i=1}^{n} r_{i,i}$$

to be the total number of impossible assignments. And define

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

to be the total number of conflicts in the current assignment. We will show that either ρ decreases with each application of a rule in $\Pi_{proto}(n)$, or it remains constant while τ decreases. Thus, (ρ, τ) will decrease lexicographically after each step for which *Active* is true. Noting that $\rho \in \{0, ..., n\}$ and $\tau \in \{0, ..., \binom{n}{2}\}$, this is equivalent to having

$$U \triangleq \left[\binom{n}{2} + 1\right] \rho + \tau$$

decrease at each step. Since $U \geq 0$, this process must eventually stop. The function U is essentially a Lyapunov function for the system in the sense of Proposition 5.2.

Lemma 6.3 *If $m > 1$ then Π_{rf} satisfies the property Safe$(m) \wedge Active \wedge U = k$* **co** *$U < k$.*

Using Lemma 6.3 and Proposition 5.2 we can, finally, show the following.

Theorem 6.3 *For all $m > 0$, Π_{rf} satisfies the property Safe$(m) \wedge U = m \mapsto \neg Active$.*

7 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), basic expressions on these types, and *programs*. A program is essentially a specification as defined in Section 4 except that rules are of the form

$$\begin{aligned} x_1 &:= expr_1, \\ &\vdots \\ x_k &:= expr_k \end{aligned}$$

– that is, they are sets of assignments (as opposed to arbitrary relations as in Section 4). The specification of programs in CCLI has two other features. First, programs can be parameterized (as in $\Pi(k)$) and instantiated later (as in $\Pi(0.1)$). Second, programs may be composed with *hidden* and *shared* variables as in

$$\Pi_1 \circ \Pi_2 \textit{ sharing } x, y$$

which means that any use of variables named x or y in Π_1 and Π_2 refer to the same object while all other variable references are local to Π_1 or Π_2 as the case may be.

After defining any number of variables and parameterized programs, execution is initiated by calling *exec* Π on a program with no parameters. Then, every 10-100

ms (depending on the configuration), CCLi executes all the clauses in Π. 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 can be found at the URL

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

8 Conclusion

We have introduced a 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 build, involving both control (albeit 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 actual vehicles running verified CCL programs.

More generally, we are examining the formal properties of CCL especially composition with respect to automated reasoning tools, local and shared variables, refinement and refinement operators, and alternative asynchronous semantics that guarantee that each clause is executed with a specified frequency.

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. The idea for a CCL interpreter came from discussions between the author and Uluç Saranli. Natarajan Shankar made several suggestions regarding self-stabilizing protocols. The author also thanks the *CCL Courselet* attendees for serving as beta testers for the CCL software tools. This research is supported in part by the AFOSR, grant number F49620-01-1-0361.

References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications 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 *Conference on Decision and Control*, Las Vegas, NV, 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. Submitted for review.

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

[7] 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.

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

[9] E. Klavins. Automatic synthesis of controllers for distributed assembly and formation forming. In *Proceedings of the IEEE Conference on Robotics and Automation*, Washington DC, 2002.

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

[11] E. Klavins and U. Saranli. Object oriented state machines. *Embedded Systems Magazine*, pages 30–42, May 2002.

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

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

[14] P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In *3rd International Workshop on Rewriting Logic and its Applications*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[15] L. C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, July 2000.

[16] U. Saranli, M. Buehler, and D.E. Koditschek. RHex: A simple and highly mobile hexapod robot. *International Journal of Robotics Research*, 20(7):616–631, July 2001.

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