A Language for Modeling and Programming Cooperative Control Systems

Eric Klavins
Electrical Engineering Department
University of Washington
Seattle, WA 98195
klavins@washington.edu

***Abstract*— In this paper we describe a software tool called CCLi (for CCL interpreter) that implements the *Computation and Control Language* (CCL). CCL is a language for modeling and programming of robotic and control systems. CCLi is used to simulate CCL models and programs and can also be used to execute CCL programs on actual robots. The language is particularly well suited to concurrent, partially synchronized processes interacting via communications and the environment, such as would describe cooperative control tasks. This paper describes the syntax and semantics of the language and gives examples of its use in modeling and programming cooperative control and multi-robot systems.**

I. Introduction

We are interested in engineering algorithms and software for cooperative control systems, which consist of possibly large numbers of vehicles or robots. In cooperative control, we usually assume that robots are coupled to each other via communication and that they are embedded in a common, physical, environment. For these systems, the algorithm design process ideally involves a rigorous analysis of the properties (stability, robustness, performance) of candidate algorithms. Meanwhile, the software design process usually involves debugging, testing and simulating an implementation of a verified algorithm. Often, there is a disconnect between these two aspects of system that delays the development of working prototypes and introduces bugs and complexity into the system.

One way to bridge this gap is to write control algorithms and environment models in the same language as the implementation of the control algorithms. Algorithms so written could then be analyzed using the operational semantics of the language, and the same code that is analyzed can be executed on actual hardware, or used in simulation and testing. This is, for example, the paradigm used by *Esterel* [5] for reactive system design and verification of synchronous systems.

The *Computation and Control Language* (CCL) is a modeling *and* programming language specific to the domain of distributed and possibly asynchronous control algorithms. A CCL program consists of a set of *guarded commands* each of which represents a possible action of the system. One supposes that the each processor (robot, vehicle,...) *owns* a number of guarded commands defining the processor's program. The environment may also be modeled by a set of guarded commands that define (a discrete-time model of) the physical dynamics of the system. Concurrency is achieved by allowing the execution of the commands to be interleaved in various different ways. In particular, execution may follow a number of different schedules (or, equivalently, obey certain fairness constraints). The most relaxed schedule requires only that all guarded commands be executed infinitely often. A more restrictive schedule requires that each guarded command be executed at essentially the same frequency.

CCLi allows us to write down concurrent, guarded command programs; compose programs into larger programs; and execute the programs with various scheduling options. CCLi is *type-safe* [14], so that incorrectly typed programs are identified at compile time. It provides a number of libraries containing math, graphics, interprocess communications and TCP/IP functionality. Finally, the syntax of the language closely follows the formal semantics of CCL [10]. Thus, CCLi programs bear a strong resemblance to formal notation for concurrent programs. In future work, we plan to exploit this resemblance by encoding the semantics of CCLi in an automated theorem proving environment [3].

The contribution of this paper is the introduction of CCLi. In Section III, we describe the syntax of CCLi, the CCLi compositional operators, how to schedule CCLi programs, and the interface between CCLi and other programming languages and libraries. In Section IV, we give a number of examples of the use of CCLi to model cooperative control systems. In Section V, we describe the formal semantics of CCLi.

II. Related Work

Much of the work in robot programming languages focuses on making robots easy to program by defining a library of commonly used functions, or syntax especially suited for robotics. Many such efforts are oriented toward seamlessly combining descriptions of high level tasks (such as patrolling a room) with intermediate and low level code (such as interfacing with device drivers) [15]. In contrast, CCLi is intended to be as much a *notation* for modeling decentralized robotic or control systems as it is a language for simulating and programming robots. In this sense, CCLi is closer to Esterel [5] or Charon [4], in that the goal is to write down small models of systems and use them

for stability and performance analysis, simulation, control and possibly, because the operational semantics of these languages is relatively simple, verification.

While many such modeling languages are based on hybrid state machines, CCLi is based on an interleaving model of commands appropriate for decentralized, asynchronous systems. In appearance CCLi is much like UNITY [6], which is a language for writing parallel algorithms and, when interprocess communication is used, I/O Automata [12]. In fact, an implementation of UNITY has been explored, called *ImpUNITY* [16], that is somewhat similar to CCLi. The difference between CCLi and UNITY or I/O Automata is CCLi's focus on alternative scheduling options, program composition and an easy interface to other programming languages.

The semantics of CCLi, and the formal reasoning they enable, are based on transition systems and temporal logic [11], [13]. As in the *Simple Programming Language* [13], we write $P \models \phi$ to mean that all the behaviors allowed by the program P satisfy the temporal logic formula ϕ.

III. CCLi Syntax and Scheduling

In this section we describe most of the basic syntax of CCLi. The reader should consult the CCLi manual [1] for complete details. In particular, various shorthand expressions and syntactic sugar are not covered here, as well as certain useful operators. The first sub-section, on basic expressions, is meant only to aid in presentation. The low-level syntax of CCLi is not necessarily new nor especially unique. However, the manipulation of programs described in the Section III-C is one of the main features of CCLi, and requires some familiarity with the basic expression syntax of CCLi.

A. Basic Expressions

CCLi supports values the with following *atomic* types: `unit`, `boolean`, `integer`, `real` (floating point) and `string`. Compound types can be lists, records, lambda abstractions and external functions. In CCLi, variables are assigned a type by their first usage. For example, the assignments

```
x := { 1, 2, 3 };
y := [ a := "ccl", b := true, c := 3.14 ];
```

initialize the variable `x` to be of type `integer list` and `y` to be a record of type `[ a : string, b : boolean, c : real ]`. Future occurrences of `x` and `y` must be consistent with these initial assignments, or CCLi reports an error and aborts.

Expressions in CCLi may use any of the common arithmetic operations, comparisons and boolean connectives. Strings may be concatenated using the `<>` operator. An element may be prepended to a list using the `@` operator and lists may be concatenated using the `#` operator. An element of a list `L` may be extracted, as in `L[5]`, as though the list were an array, although the user must check for out-of-bounds conditions to avoid a run-time error. For example, the following expressions evaluate to `true`:

```
"a" <> "b" = "b";
1 @ { 2, 3 } = { 1 } # { 2, 3 };
```

Here, "`=`" is the equality operator in CCLi.

Because of the frequency with which one would like to obtain the recent history of a variable in control code, CCLi makes available the previous value `'v` of all variables `v`. Thus, one may write

```
t := t + 't
```

which, loosely, corresponds to the difference equation $t_{k+1} = t_k + t_{k-1}$. In general, one may similarly "prevify" an entire expression e, which is syntactic sugar for changing all free occurrences of variables `v` in e with `'v`.

If `b` is a boolean expression and `e1` and `e2` are expressions of type T, then

```
if b then e1 else e2 end
```

is an expression of type T that evaluates to `e1` if `b` evaluates to `true` and `e2` otherwise. If `e1` is an expression of type T_1 and `e2` is an expression of type T_2 that may contain free occurrences of a variable `x` (occurring with type T_1), then

```
let x := e1 in e2 end
```

is an expression of type T_2 that evaluates to `e2` with `e1` substituted for `x` in `e2`. This allows the programmer to declare local variables.

Functions in CCLi have two types: Lambda abstractions and external functions. If `e` is an expression of type T_1 and `x` occurs free in `e` with type T_2, then

```
lambda x . e
```

is an expression of type $T_2 \rightarrow T_1$. The above expression may be applied to an expression f as in

```
( lambda x . e ) f
```

to yield the expression `e` with all free occurrences of `x` in `e` replaced by `f` (as in standard lambda calculus [8]). Lambda expressions are polymorphic. Thus, if `f` is declared

```
f := lambda x . lambda y . x @ y
```

then `f` has type $T \rightarrow T\ list \rightarrow T\ list$. For example, with the above definition of `f`, the following expressions are both legal

```
a := f 1 {};
b := f true {};
```

Lambda abstractions may also be declared with the `fun` operator, which allows them to be recursive. For example, the factorial function is defined by

```
fun f x . if x = 1
            then 1
            else x * f (x-1)
          end;
```

Remark: There are no pointers in CCLi and all functions are *call-by-value*. Thus, memory is allocated and freed during every evaluation, thereby avoiding periodic garbage collection episodes that interfere with real-time execution.

External functions are declared as in

```
external [ x := int, ... ] f ( 'a list )
  "library.so" "func";
```

which declares an external function `f` to take a list of any type (`'a` is a type variable) and return a record that has an integer field `x` and any number of other fields of arbitrary type. The implementation of `f` in this case should be in the shared object library named `library.so` and should be called `func`. External functions are how CCLi programs interface to the outside world. For example, there is a library of external functions for showing graphics on the screen, for communicating via UDP, and so on. We have also implemented an interface to the Caltech Multi-Vehicle Wireless Testbed [7] hardware, allowing CCLi programs to send commands to vehicle effectors and receive input from vehicle sensors. The CCLi manual [1] explains the more of the details of external functions and their calling conventions.

Remark: CCLi is a *strongly typed* language and, as was noted above, CCLi infers the type of a variable by its initial assignment. CCLi performs type inference [14] on all expressions (and guarded commands and programs, as defined in the sequel), and reports an error before executing any program, if an expression has a type inconsistency. The benefit of type checking and inference is that programmers do not need to separately define the types of variables (as in C-like languages) and, more importantly, type checking catches a great number of initial coding errors. Type checking and inference is, thus, a form of verification (of weak, but important, properties). This is, for example, very useful when composing CCLi programs, as described below.

B. Guarded Commands

A *guarded command* is an expression that can only appear inside the scope of a program (see Section III-C). Such an expression consists of a boolean *guard* and a list of assignments making up the *command*. For example,

```
x > 0 : {
          x := -x,
          mode := "waiting"
        }
```

is a guarded command that tests whether the value of `x` is positive, and if it is, re-assigns the variables `x` and `mode` according to the list of assignments inside the brackets. If the value of `x` is not positive, then the assignments within the command are not executed.

The intent is that the guarded command gets executed, along with other guarded commands, repeatedly in a CCLi program. The system state, defined to be the current value of all variables, is thus tested by the guard and certain variables updated if the test comes out true. In Section V on the operational semantics of CCLi, this notion is made formal.

C. Programs and Composition

A *program* in CCL consists of (1) a set of variable initializations and (2) a set of guarded commands. Programs may be declared with parameters. For example,

```
program p(x0,k) := {
  x := x0,
  x < 0.0  : { x := x + k },
  x >= 0.0 : { x := x - k }
};
```

is a simple program with two parameters, `x0` and `k`, and one local variable `x`. CCli infers the types of the parameters to be `real` since that is how they are used. The general syntax for *atomic* programs is:

atomic_prog ::= { *statement_list* }
statement ::= *initializer* | *needs_decl* | *command*
initializer ::= *var* `:=` *expr* `;`
needs_decl ::= `needs` var_1, ..., var_n `;`

where *expr* refers to a basic CCLi expression as described in Section III-A and *command* refers to a guarded command as defined in Section III-B. The `needs` keyword is used to declare a list of variables as local without initializing them. Their types will be inferred from how they are used in the rest of the program and their values must be initialized elsewhere by some other program (see below).

Programs are interpreted as follows:

1) All variables are initialized according to their initial assignments;
2) The guarded commands are executed repeatedly according to some schedule (see Section III-D).

Thus, a program *defines a dynamical system* that runs forever, and not a sequence of statements that are executed once each (this notion is defined formally in Section V). For example, if the above program `p` is instantiated as `p(10,1.25)` and the guarded commands are scheduled *in order*, then a trajectory of the system defined by `p(4,1.25)` is

step	0	1	2	3	4	5	...
x	4.0	2.75	1.5	0.25	-1.0	0.25	...

which goes on forever (until the user kills the program), settling into a limit cycle where `x` alternates between -1.0 and 0.25.

Programs may be *composed*, meaning that their variable initializations and guarded command sections are essentially concatenated (so ordering is important). A variable `v` appearing in both sub-programs of a composed program may be specified as *shared*, meaning that references to `v` in the sub-programs will be to the same object. Other variables will remain local to the sub-programs. For example, we can write a program that implements an output function $x \mapsto x/2$ for the dynamical system defined by `p`:

```
program q(h) := {
  needs x;
  y := h x;
  true : { y := h x }
};

program main() := p ( 4, 0.1 )
                + q ( lambda x . x/2 )
                  sharing x;
```

where `p` is defined as before. The program `main` has three variable declarations (one from `p` and two from `q`) and *two*

guarded commands (one each from `p` and `q`). The variable `x` appearing in `p` and `q` refers to *the same* object. Any variables with the same name in `p` and `q` but not shared (there are none in this example) would refer to different objects. For example, if `y` appeared in `p`, it would be a different object than the `y` appearing in `q` above, since `y` is not declared as shared. Program composition with sharing is defined explicitly in Section V. An example trajectory for the above program `main` is

step	0	1	2	3	4	5	...
x	4.0	2.75	1.5	0.25	-1.0	0.25	...
y	2.0	1.375	1.25	0.125	-0.5	0.125	...

Programs may be composed more generally, as in

```
program main() :=
  compose x0 in L : p ( x0, 0.1 );
```

where `L` is an arbitrary list of type `real` (in this particular case). If the list `L` has n elements, then a program `main` so defined contains n distinct copies of the variable `x`, each one initialized to one of the values in `L`. The general syntax for program composition is

prog ::= *atomic_prog* | *inst* | *composite* | `(` *prog* `)`
inst ::= *var* `(` $expr_1$, ..., $expr_k$ `)`
composite ::= *binary_comp* | *nfold_comp*
binary_comp ::= *prog* `+` *prog sharing* var_1,...,var_k
nfold_comp ::= `compose` *var* `in` *expr* `:` *inst*

The CCLi type checker and inference engine checks that the types of shared variables match, that parameters are instantiated correctly and so on, producing useful compile-time error messages if a type inconsistency is found. More examples of program composition appear in Section IV.

D. Scheduling and Execution

Programs are written in ASCII files, which may `include` other files in the standard way. Executing "`ccli file.ccl`" causes CCLi to parse the expressions in `file.ccl`. When parsing is complete and no type errors have been found, CCLi will look for a program with no parameters called `main` in the CCLi symbol table, initialize the variables in `main`, and begin executing the guarded commands in `main` and its subprograms.

As we have noted, the execution of guarded commands can be scheduled in various ways during execution. The most simple schedule is *in-order* scheduling, and is the default method for CCLi. CCLi has two other ways of scheduling: The first is called *epoch* (`ccli -r`) and the second is *unity* (`ccli -u`).

In *in-order* scheduling guarded commands in `main` are executing in order of appearance (which is why order may be important when composing programs).

In *epoch* scheduling, execution is segmented into "epochs". In each epoch, each guarded command is executed exactly once. The order in which commands are executed in any given epoch is (pseudo)random. The intent behind *epoch* scheduling is to mimic the possible interleavings of commands supposing that they were executing on different processors running at essentially the same rate.

In *unity* scheduling, a guarded command in `main` is chosen (pseudo)randomly at each step and executed. There is no guarantee that at any point each clause will eventually be executed (as is required by the formal UNITY fairness constraint [10]), but given the state of the art of pseudo-random number generators, it is a virtual certainty that they will. The intent behind *unity* scheduling is to mimic the possible interleavings of commands supposing that they are executing on different processors each running at totally random and uncorrelated rates.

E. CCLi Libraries

CCLi provides libraries of convenience functions as well as for interfacing CCLi with the real world. We describe these briefly here, the reader is directed to the CCLi manual [1] for details.

Standard Libraries: There are libraries of functions for printing to the screen, reading console input, common mathematical functions, and operating on lists. Also provided are timers (in seconds, milliseconds and microseconds) useful for real time code. Some of these are defined in CCLi as lambda abstractions, and others as external functions. Several libraries for other functionality (such as a numerical integrator or trajectory generator) are in development, but not yet included in the CCLi distribution.

Graphics: There is a library for displaying simple graphics (such as simulation data) that is essentially a wrapper around the GDK/GTK open source graphics libraries [2].

Interprocess Communications: Sub-programs may communicate via "mailboxes" using CCLi's `iproc.ccl` library. It provides three functions: `send`, `recv` and `inbox`. The first, `send`, is used to send a message (of any CCLi type) to the mailbox (a simple queue) of another process. Mailboxes are indexed by integer ID numbers. The second function, `recv`, is used to pop the first message off the top of a given mailbox. `recv` is non-blocking if the intended mailbox is not empty. The last function, `inbox`, returns `false` if the designated mailbox is empty, and `true` otherwise. In Section IV, we give an example of the use of this library.

TCP/IP: Multiple instances of CCLi running on separate machines connected via TCP/IP may communicate via UDP datagrams. Messages of any type may be sent (something that CCLi can not type check, unfortunately) and, as is usual with UDP, may or may not arrive at their destination. If messages do arrive, they may arrive in or out of order. The interface to the message passing functions is virtually the same as with the interprocess communications library already described. Thus, it is easy to prototype concurrent sequential code in a single process and then "distribute" it to multiple processors without changing much.

Hardware Interfaces: The distribution of CCLi includes our own library for interfacing with the Caltech Multi-Vehicle Wireless Testbed [7] hardware. The basic

methodology is to wrap an external CCLi function around C++ code that interfaces with device drivers for sensors and effectors, and is very straightforward. Although the MVWT library released with the distribution may not be directly useful for other users, it can serve as an example of how to interface CCLi with actual hardware.

IV. An Extended Example

In this section we describe a complete example. The system we consider is a scheme for robot "flocking". Recently, a rigorous analysis [9] of this system has spurred interest in similar algorithms for decentralized robot tasks, wherein each robot *averages* some value obtained from its neighbors and uses that average in its own control. In flocking, a robot with position $X_i(k) \in \mathbb{R}^2$ and heading $\theta_i(k) \in [0, 2\pi)$ at step k uses the average heading of its neighbors to adjust its own according to the rule

$$\theta_i(k+1) = \frac{1}{|N_i(k)|} \sum_{j \in N_i} \theta_j(k) \qquad (1)$$

where $N_i(k)$ is the set of neighbors of robot i, defined by

$$N_i(k) \triangleq \{j \mid ||X_i(k) - X_j(k)|| \leq r\}$$

with r being the maximum distance a message can be communicated (due, say, to power constraints). The position of robot i is adjusted by moving it in the direction of its heading, as in

$$X_i(k+1) := X_i(k) + \delta(\cos\theta_i(k), \sin\theta_i(k)). \qquad (2)$$

A. Flocking in CCLi

From a distributed systems perspective, the algorithm exactly as described above is not well decentralized, requiring, as it does, some way of synchronizing the robots. That is, they must first send their data, then receive all of their neighbors data and then move. However, the algorithm can be used as the basis of a more decentralized version.

We first define a function that filters incoming data according to whether it is coming from a nearby robot or not:

```
fun recv_filter r i x N .
  let m := recv ( i ) in
    if dist x (m.X) <= r
      then replace N (m.from) (m.a)
      else N
    end
  end;
```

Here, the `recv` command returns a record of type

```
[ to : integer, from : integer,
  a : real, X: real list ]
```

containing the receiver and sender IDs, the heading (`m.a`) of the sender and the position (`m.X`) of the sender. The parameter `N` is a vector (actually a CCLi list) of known headings of all robots. The `recv_filter` function checks whether the distance between the receiver `x` and the sender `m.X` is less than `r`. If it is, it replaces the appropriate component of `N` with the received heading. Otherwise, no update is made.

Next, we define a program that receives heading data from a robot's neighbors. The program also initializes the position, orientation and identity of the robot (passed as parameters `X0`, `a` and `i`).

```
program receiver ( X0, a0, i, n, r ) := {
  X := X0;
  a := a0;
  N := makelist n 0.0;
  inbox ( i ) : {
    N := recv_filter r i X N
  }
}
```

The single guarded command simply checks, over and over, whether the robot has received a message. If it has, it attempts to update the neighbor vector. It may seem strange that the robot has to enforce the distance constraint in the communications scheme. This is a modeling detail, however, that could be handled in several other more or less satisfactory ways.

We next define a program that sends data to other robots. It takes as a parameter a number `delta` that defines how often it sends its heading. Each time it sends, it sends to a different robot `j`, thereby attempting to send its heading to any particular robot approximately every `n * delta` seconds. The `dclock()` external function gives the number of seconds (as a `real`, accurate to the microsecond) since the process started.

```
program sender ( i, n, delta ) := {
  needs X, a;
  t := dclock();
  j := 0;
  dclock() - t > delta : {
    send ( [ to := j, from := i,
             a := a, X := X ] ),
    t := dclock(),
    j := ( j + 1 ) mod n
  }
}
```

Finally, we define a program that periodically updates the robot's heading and position. It also clears the neighbor heading vector for the next round of communication. Note that we are assuming that no robot has heading identically zero, as we are using that as a placeholder for a robot that has not communicated its heading.

```
program mover ( delta ) := {
  needs X, a, N, t;
  dclock() - t >= delta : {
    a := update_heading a N,
    X := update_pos delta X a,
    N := smult 0.0 N,
    t := dclock()
  }
}
```

The functions `update_heading` and `update_pos` are CCLi encodings of Equations (1) and (2) and are not listed for the sake of brevity.

The program for a single robot is then

```
program robot ( X0, a0, i, n, r, d1, d2 ) :=
    receiver ( X0, a0, i, n, r )
  + mover ( d1 ) sharing X, a, N, t
  + sender ( i, n, d2 ) sharing X, a;
```

and the program for an entire flock of robots is

```
program main() := compose i in range n :
   robot ( Xlist, alist, i, n, r, d1, d2 )
```

We assume that the initial conditions (`Xlist` and `alist`), the number `n` of agents and the parameters `r`, `d1`, `d2` have been defined (they could be read from the command line, for example). If these program definitions and the supporting code are contained in a file called `flock.ccl`, then executing, for example,

```
ccli flock.ccl -r
```

will initialize the variables of the system and then repeatedly execute the $3n$ guarded commands in the `main` program in random order. Other programs could be composed with the above to, for example, print data or to create an animation of the system. See the distribution code [1] for the complete code of `flock.ccl` and some variations.

B. Experimenting with `flock.ccl`

The various parameters affecting the performance of the algorithm are `r`, the radius of communication, `d1`, the amount of time between updates to the position `X` and heading `a`, and `d2` the amount of time between transmissions of the robot's heading. Figure 1 shows three data sets obtained by running `flock.ccl` with different values for these parameters, and for 20 robots with random initial conditions.

Several phenomena can be observed in the data. First, as is expounded upon in analysis of the flocking algorithm [9], the radius affects the connectivity of the communication graph and therefore number of unique headings eventually found by the group. In addition, by describing a more distributed realization of the flocking algorithm, we observe other behaviors. For example, the ratio between `d1` and `d2` affects the convergence time of the algorithm — if it converges at all. This is because, if `d2` is not small compared to `d1`, the individual robots may update their headings before receiving information from all of their neighbors. Of course, if `d2` is small enough, this is not a problem. But a small `d2` requires more bandwidth, which may be at a premium as the number of robots increases.

We also notice from the data that the robots become less synchronized as the simulation progresses: Initially the *(time,heading)* pairs form columns, and later they are more spread out. The frequency with which the robots is still essentially the same, but offsets drift.

Bringing to light phenomena similar to the above, that arise from the decentralization of an algorithm (even one that supposedly *is* decentralized), is one of the main advantages of CCLi.

Furthermore, the code above could be easily extended to run on actual robots, if a suitable interface to the sensors, effectors and communication hardware were defined (e.g. as external functions) to replace the `iproc.ccl` library and the assignments in the `mover` sub-program.

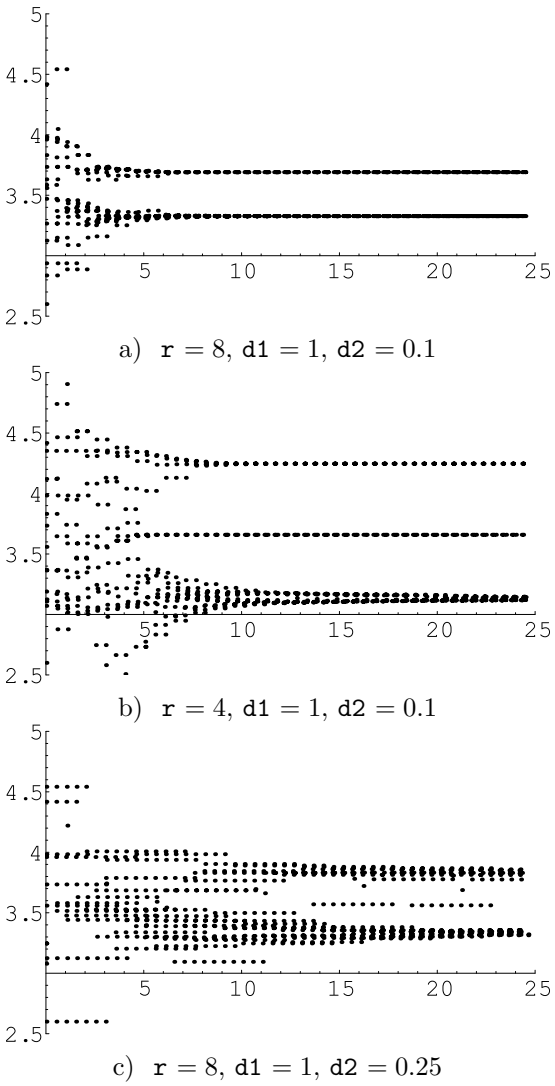

Fig. 1. Data obtained from executing `flock.ccl` as defined in Section IV-A. Each point represents a *(time,heading)* pair, reported by inserting a `print` statement into the program `mover`. In each data set, 20 robots were initially randomly positioned in a 20×20 m square. (a) and (b) demonstrate that the communication radius affects the number of connected groups, each moving along a common heading. (a) and (c) demonstrate that sending data less frequently compared to the update time in the `mover` program lengthens the convergence time of the algorithm, as in the last data set.

V. CCLi Semantics

The goal is to make CCLi programs be essentially the formal notation for distributed dynamical systems. Thus, the operational semantics of CCLi need to be precisely defined. By operational semantics, we mean *the interpretation of a static object as a trajectory (or behavior).* For example, the equation $\dot{x} = -ax$ defines an object in the language of differential equations whose operational semantics is a flow (set of solutions of the equation) induced on $\mathbb{R}$. In this section, we briefly outline how to define the semantics of CCLi, focusing on the unique aspects of CCLi (i.e. those not found in CCL). A more complete treatment of CCLi can be found elsewhere [10].

A. States and Actions

We let V be the set of all *variable symbols* and *Val* be the set of values that the variables may take under the

CCLi typing system. Since CCLi type checks programs, we can suppose that the type of a variable does not change, even though its value might. The following definitions are similar to those found in the temporal logic literature [11].

Definition 5.1: A **state** s is a function from V to Val. The value of a particular variable $v \in V$ with respect to a state s is denoted $s[\![v]\!]$. The set of all states in denoted by S.

Definition 5.2: The **meaning** of a CCLi expression f, denoted $[\![f]\!]$, is a function from states into values and is defined to be the value obtained by replacing all (free occurrences of) variables in f by their values under the state s and then evaluating all lambda expressions using β-reduction.

Definition 5.3: An **action** is a boolean valued expression over variables in V, primed variables in V' and constant symbols.

The only actions in CCLi are assignments and guarded commands. The assignment `x := y` in CCLi is equivalent to the *action* $x' = y$ in the above. Guarded commands are discussed below.

Definition 5.4: The **meaning** of a an action a, denoted $[\![a]\!]$, is a function from $S \times S$ into values and is defined to be 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 and then evaluating all lambda expressions (and external functions).

Definition 5.5: A **guarded command** is a pair (g, r), usually written $g : r$, where g is a predicate and r is an action (a list of assignments in CCLi). 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

$$true : \{\} \tag{3}$$

Definition 5.6: A **program** 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 **simple composition** is $P_1 \circ P_2 = (I_1 \wedge I_2, C_1 \cup C_2)$.

In a CCLi program, the initial condition is obtained by evaluating all variable initializations in the program.

Definition 5.7: If $P = (I, C)$ is a program and $U \subseteq V$ is a set of variable symbols, the program obtained by **hiding** the variables in U is defined by

$$hide(P, U) = P(\forall v \in U : P.v \ / \ v, \ P.v' \ / \ v'),$$

that is, the program obtained by replacing all free occurrences $v \in U$ in either I or a command in C with the new variable symbol $P.v$.

Definition 5.8: If P_1 and P_2 are programs and U is a set of variables, then their **shared variable composition** is

$$P_1 \bullet P_2 \ sharing \ U \triangleq$$

$$hide(P_1, V(P_1) - U) \circ hide(P_2, V(P_2) - U).$$

Thus, a variable v not in U but appearing in both P_1 and P_2 is renamed to $P_1.v$ within P_1 and to $P_2.v$ within P_2 – that is, it is local to those programs. On the other hand, if $v \in U$, then the symbols u and u' in P_1 and P_2 refer to the same object in $P_1 \bullet P_2$ *sharing* U. In this manner we see that program composition, as discussed in Section III, is concatenation of programs, with non-shared variables being renamed.

B. Behaviors and Schedules

Definition 5.9: 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 $\omega : \mathbb{N} \to C \cup \{skip\}$ that assigns to each step either a guarded command from C or the command *skip* of Equation (3). The behavior σ is an **ω-behavior** for the program $P = (I, C)$ if

1) $\sigma_0[\![I]\!]$
2) $\forall k \ . \ \sigma_k[\![\omega(k)]\!]\sigma_{k+1}$.

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

Based upon these definitions, we will provide several scheduling tactics that define what it means for a behavior σ to satisfy a program P. Each scheduling approach is defined by a *fairness constraint*, that limits how often each guarded command may be selected for execution. For each constraint M we will define the M-meaning of P, denoted

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

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 default schedule for CCLi is to execute the guarded commands *in order*. This is defined formally by:

Definition 5.10: (ORDER FAIRNESS) Given a program $P = (I, C)$ with $C = \{c_0, c_1, ..., c_{n-1}\}$ considered as an ordered set, and a behavior σ, then $\sigma[\![P]\!]_{ORDER} = true$ if and only if there exists a schedule ω such that

1) σ is an ω-behavior for P
2) $\omega(k) = c_{k \bmod n}$.

On the other hand, the least restrictive schedule for a program is one that eventually applies each guarded command, interleaving them in any order. This schedule can be approximated by CCLi using the `-u` flag. This is essentially the same as the semantics of UNITY [6]. The fact that each guarded command is eventually applied is equivalent to saying that each command enjoys *weak fairness*.

Definition 5.11: (UNITY FAIRNESS) Given a program $P = (I, C)$ and a behavior σ, then $\sigma[\![P]\!]_{UNITY} = true$ if and only if there exists a schedule ω such that

1) σ is an ω-behavior for P
2) for all $c \in C$, the set $\omega^{-1}(c)$ is infinite.

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

1) σ is an ω-behavior for P
2) 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

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

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

Remark: In the programs defined in Section IV we made heavy use of the `dclock()` function, which in the implementation of CCLi, gives the system time as the number of seconds since the CCLi process was started. Technically, this is a function outside of the operational semantics as we have described here. One can approximate the meaning of the `dclock()` function by (roughly) defining a global time variable T and appending the (nondeterministic) action $T' \in (T, T + \delta)$ to the rule in each guarded command.

In other work on CCL [10], we define for each fairness constraint M, the *models* relation $P \models_M \phi$, where P is a CCL program and ϕ is a formula in temporal logic [11]. We say that $P \models_M \phi$ is true if and only if every behavior allowed by P satisfies the formula ϕ. We further define *inference rules* of the form: To prove that $P \models_M \phi$, it suffices to prove that $P \models_M \psi_i$ for $i = 1, ..., k$ where each ψ_i is a simpler formula than ϕ. Such rules are the basis of mathematical reasoning and provide a basis, introduced in other papers [10], for determining what properties a given program enjoys. When carefully formulated, a complete set of inference rules can be encoded into an automatic theorem prover to help automate the verification process. We have begun to extend the work by Paulson on automating temporal logic in *Isabelle* [3] to reason about CCLi. We hope to report on this in future work.

VI. Discussion

CCLi represents an attempt to encode the CCL formalism in a usable piece of software. It allows researchers to quickly encode distributed and decentralized algorithms in a fashion that is at once similar if not identical to mathematical notation and executable. The compositional operators in CCLi are powerful and useful for (1) building up programs from simpler programs and (2) defining systems composed to large numbers of communicating, asynchronous processes, as the Example in Section IV shows.

Our investigation into CCL and related formalisms has so far been focused on two separate issues: defining a formalism and notation for distributed robot algorithms and defining software based upon it. Eventually we will encode CCLi into a theorem proving environment so that we may use automated tools to help prove properties of the systems we define. The goal will be to use the same syntax in the theorem prover as with executable programs. As far as the language is concerned, we are continuing to refine and expand it to make it even more useful both a research and educational setting.

Acknowledgments

This work is supported in part by AFOSR grant number F49620-01-1-0361. The author wishes to thank Jason Hickey and Mani Chandy for guidance on the structure of CCL and Steve Waydo for contributions to the CCLi code. The author also wishes to thank all the alpha testers of CCLi for their suggestions and help finding bugs.

References

[1] The CCLi web page. http://sveiks.ee.washington.edu/ccl/.
[2] The GNOME developmers site. http://developer.gnome.org/.
[3] *Isabelle: A generic theorem prover*. LNCS 828. Springer-Verlag, 1994.
[4] 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.
[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addision-Wesley, 1988.
[7] T. Chung, L. Cremean, W. Dunbar, Z. Jin, E. Klavins, D. Moore, A. Tiwari, D. van Gogh, and S. Waydo. A platform for cooperative and coordinated control of multiple vehicles: The Caltech multi-vehicle wireless testbed. In *Proceedings of the Conference on Cooperative Control and Optimization*. Kluwer, 2002.
[8] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.
[9] A. Jadbabaie, J. Lin, and A. S. Morse. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Transactions on Automatic Control*, 2002. To appear.
[10] E. Klavins. A formal model of a multi-robot control and communication task. In *42nd IEEE Conference on Decision and Control*, Maui, HI, December 2003.
[11] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
[13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
[14] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Methods and Semantics*, volume B, pages 365–458. Elsevier, 1990.
[15] I. Pembeci and G. D. Hagar. A comparative review of robot programming languages. Technical report, CIRL, Johns Hopkins University, 2002. http://www.cs.jhu.edu/CIRL/publications/.
[16] R. T. Udink and J. N. Kok. Impunity: UNITY with procedures and local variables. In *Mathematics of Program Construction*, pages 452–472, 1995.