NCS Lecture 11
Distributed Computation for Cooperative Control

Richard M. Murray (Caltech) and Erik Klavins (U. Washington)

17 March 2008

Goals:

- Describe methods for modeling and analyzing distributed protocols
- Introduce the Computation and Control Language (CCL) as an example
- Explore and analyze protocols written in CCL for cooperative contro

Reading:

- E. Klavins, "A Computation and Control Language for Multi-Vehicle Systems", *Int'l Conference on Robotics and Automation*, 2004.
- E. Klavins and R. M. Murray, "Distributed Computation for Cooperative Control", *IEEE Pervasive Computing,* 2004.

Problem Framework

Agent dynamics

$$\dot{x}^i = f^i(x^i, u^i) \quad x^i \in \mathbb{R}^n, u^i \in \mathbb{R}^m$$
$$y^i = h^i(x^i) \quad y^i \in \mathrm{SE}(3),$$

Vehicle "role"

- $\alpha \in \mathcal{A}$ encodes internal state + relationship to current task
- Transition $\alpha' = r(x, \alpha)$

Communications graph $\mathcal{G}$

- Encodes the system information flow
- Neighbor set $\mathcal{N}^i(x, \alpha)$

Task

- Encode as finite horizon optimal control

$$J = \int_0^T L(x, \alpha, u)\,dt + V(x(T), \alpha(T)),$$

- Assume task is *coupled*

Strategy

- Control action for individual agents

$$u^i = \gamma(x, \alpha) \qquad \{g^i_j(x, \alpha) : r^i_j(x, \alpha)\}$$

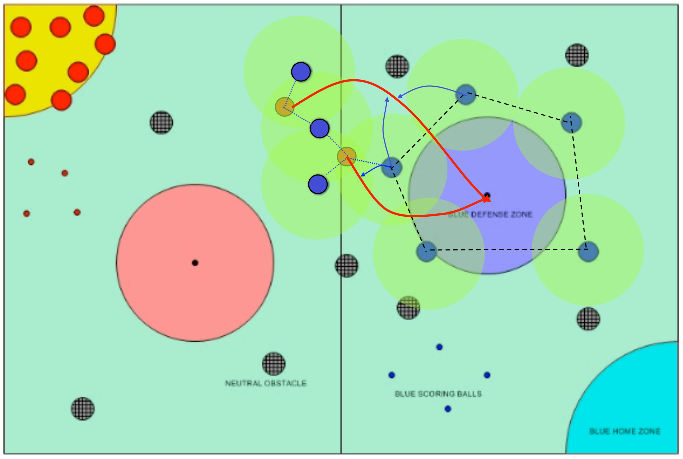
$$\alpha^{i\,\prime} = \begin{cases} r^i_j(x, \alpha) & g(x, \alpha) = \text{true} \\ \text{unchanged} & \text{otherwise.} \end{cases}$$

Decentralized strategy

$$u^i(x, \alpha) = u^i(x^i, \alpha^i, x^{-i}, \alpha^{-i})$$
$$x^{-i} = \{x^{j_1}, \ldots, x^{j_{m_i}}\}$$
$$j_k \in \mathcal{N}^i \qquad m_i = |\mathcal{N}^i|$$

- Similar structure for role update

RoboFlag Subproblems



1. Formation control
- Maintain positions to guard defense zone

2. Distributed estimation
- Fuse sensor data to determine opponent location

3. Distributed assignment
- Assign individuals to tag incoming vehicles

Desirable features for designing and verifying distributed protocols
- Controls: stability, performance, robustness
- Computer science: safety, fairness, liveness
- Real-world: delays, asynchronous executions, (information loss)

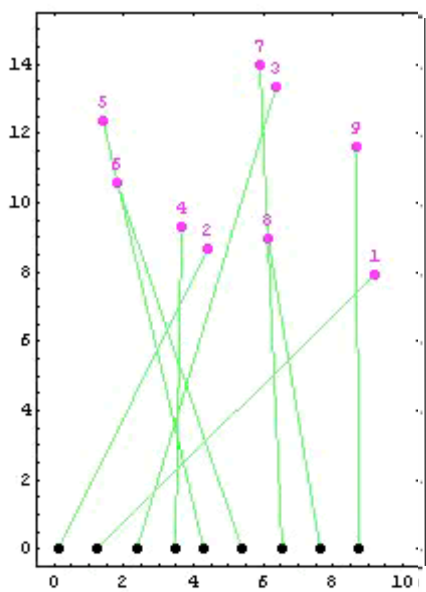
Klavins CDC, 03

Distributed Decision Making: RoboFlag Drill

Task description
- Incoming robots should be blocked by defending robots
- Incoming robots are assigned randomly to whoever is free
- Defending robots must move to block, but cannot run into or cross over others
- Allow robots to communicate with left and right neighbors and switch assignments

Goals
- Would like a provably correct, distributed protocol for solving this problem
- Should (eventually) allow for lost data, incomplete information

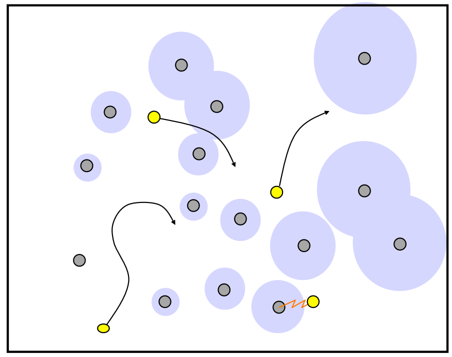
Status
- Provably correct protocol available in perfect information case, using CCL



Distributed Situational Awareness

Communications complexity

- Maintain “situational awareness”
- Assume point-to-point communications and that each robot knows its own position
- Q: how many messages are required for each robot to keep track of all other robots w/in ε?
- A: $O(n^2)$ messages (worst case)



Method #1: Distance Modulated Communication - $O(n \log n)$

- Maintain position estimates to within $kPx_i - x_jP$
- Communicate more often with robots that are closer

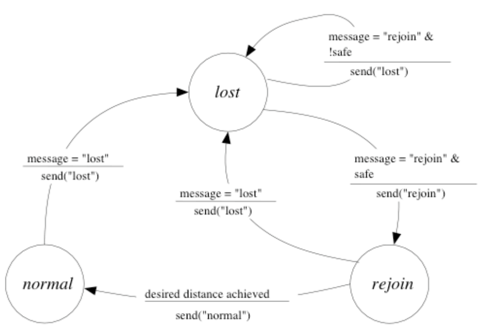
Method #2: Wandering Communication Scheme - $O(n)$

- Only moving robots need to keep track of position
- Robots transfer knowledge when they stop/start

Proof of correctness using CCL

Klavins WAFR 02

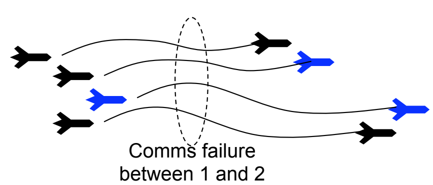
Lost Wingman Protocol Verification



Protocol specification in CCL

- Use guarded commands to implement finite state automaton
- Allows reasoning about controlled performance using semi-automated theorem proving
- Relies on Lyapunov certificates to provide information about controlled system

Lost wingman in fingertip formation



Temporal logic specification

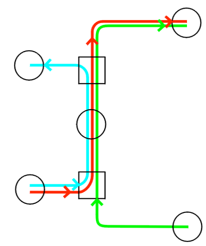
$$\Psi_l \triangleq mode = lost \rightsquigarrow \Box d(\mathbf{x}_l, \mathbf{x}_f) > d_{sep}$$

- “Lost mode leads to the distance between the aircraft always being larger than d_{sep}”

Models of Concurrency

Petri Nets and Processes

- Standard tool in Manufacturing



Hybrid Automata (Henzinger, 1996)

- Use FSM for discrete states, with dynamic inclusions in each "mode" and transitions between states

I/O Automata [Lynch: Book 1996]

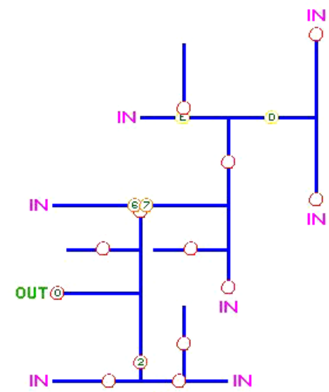
- Composition with internal / input / output actions
- Hybrid version is "sophisticated" [Lynch, Segala, Vaandrager, Weinberg: HSIII 1996]

UNITY [Chandy & Misra: Book 1988]

- Interleaving-based parallel programming
- Based on guarded commands [Dijkstra: 1975]
- Uses temporal logic for verification

Temporal Logic of Actions [Lamport: TPLS 1994]

- TL is used for specification and "implementation"
- Sophisticated treatment of fairness constraints



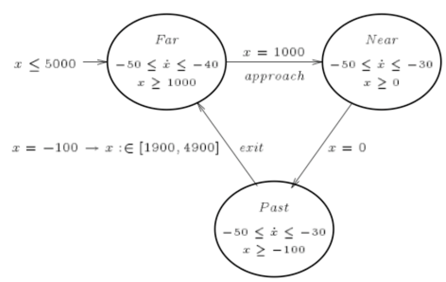
Hybrid Automata (Henzinger, 1996)

Description

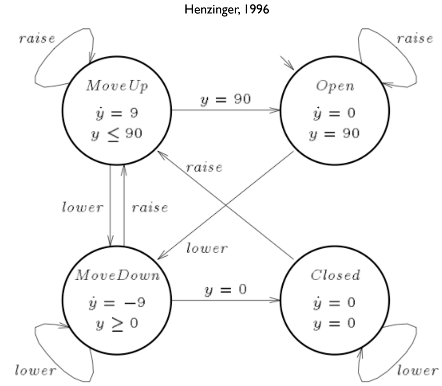
- Discrete states model different "modes" of operation
- Continuous variables within each state describe how the system evolves in that state. Allow differential inclusions: $-50 \leq \dot{x} \leq -40$
- Transitions consist of *guards* and *rules*: when a guard is true, execute the rule and then transition to a new state
- Allow composition by taking cross product of states; any transitions that have the same label must share states

Properties

- Can be used to model broad variety of systems
- Composition can lead to very large state space (n robots with r states each gives r^n states)
- Awkward to reason about in many applications (not enough structure)



Henzinger, 1996



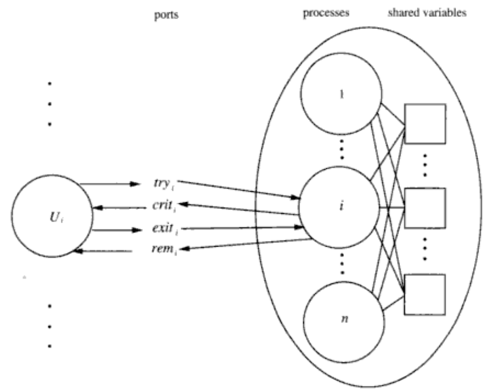
I/O Automata (Lynch, 1989)

Description

- Individual components modeled as an automaton, but with possibly infinite number of states
- Actions (transitions) are either *input, output,* or *internal*
- Composition occurs by connecting inputs to outputs (labels must match)
- Executions are given by sequence of actions; output actions trigger input actions
- Fairness constraint: each process must be allowed to execute a non-input action infinitely often in any execution => interleaving

Hybrid I/O Automata

- Add continuous dynamics via differential eqns
- Continuous execution is "interrupted" by events to give trajectories (traces)



Properties

- Extensive use in distributed algorithms
- Can reason about whether a property is true for all possible executions, which allows asynchrony of individual events

Temporal Logic

Description

- State of the system is a snapshot of values of all variables
- Reason about *behaviors* σ: sequence of states of the system
- No strict notion of time, just ordering of events
- *Actions* are relations between states: state *s* is related to state *t* by action *a* if *a* takes *s* to *t* (via prime notation: x' = x + 1)
- *Formulas* (specifications) describe the set of allowable behaviors
- Safety specification: what actions are allowed
- Fairness specification: when can a component take an action (eg, infinitely often)

Example

- Action: $a \equiv x' = x + 1$
- Behavior: $\sigma \equiv x := 1, x := 2, x := 3, \ldots$
- Safety: $\Box x > 0$ (true for this behavior)
- Fairness: $\Box(x' = x + 1 \vee x' = x) \wedge \Box\Diamond (x' \neq x)$

- $\Box p \equiv$ **always** *p* (invariance)
- $\Diamond p \equiv$ **eventually** *p* (guarantee)
- $p \rightarrow \Diamond q \equiv$ *p* **implies eventually** *q* (response)
- $p \rightarrow q\, \mathcal{U}\, r \equiv$ *p* **implies** *q* **until** *r* (precedence)
- $\Box\Diamond p \equiv$ **always eventually** *p* (progress)
- $\Diamond\Box p \equiv$ **eventually always** *p* (stability)
- $\Diamond p \rightarrow \Diamond q \equiv$ **eventually** *p* **implies eventually** *q* (correlation)

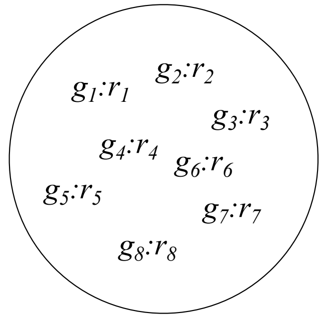
Properties

- Can reason about time by adding "time variables" (t' = t + 1)
- Specifications and proofs can be difficult to interpret by hand, but computer tools existing (eg, TLC, Isabelle, PVS, etc)

UNITY (Chandy and Misra)

Description

- Specification consists of a set of (possibly gaurded) variable assignments
- Behaviors are generated by starting an an initial state, then choosing any assignment for which the guard is true
- Command (g:r) may be evaluated in any order, at any time
- Require that all assignments be applied infinitely often in any execution (built in fairness)
- Reason about "programs" using temporal logic



Properties

- Useful for reasoning about systems in which there is very asynchronous behavior
- Fairness constraint is a bit too loose for control applications; only assume that each command executes *eventually* (instead of once every iteration)

CCL: Computation and Control Language

Formal Language for Provably Correct Control Protocols

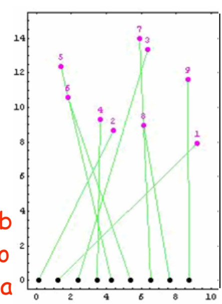
```
P(k1,k2) := {
  initializers
  guard1:rule1
  guard2:rule2
  ...
}

S(k1,k2):=P(k1,k2)+C(k1+1) sharing y,u
```

"soup" of guarded commands

composition = union

non-shared variab remain local to component progra



CCL Protocol for Decentralized Target Allocation

<u>CCL Interpreter</u>

Formal programming language for control and computation. Interfaces with libraries in other languages.

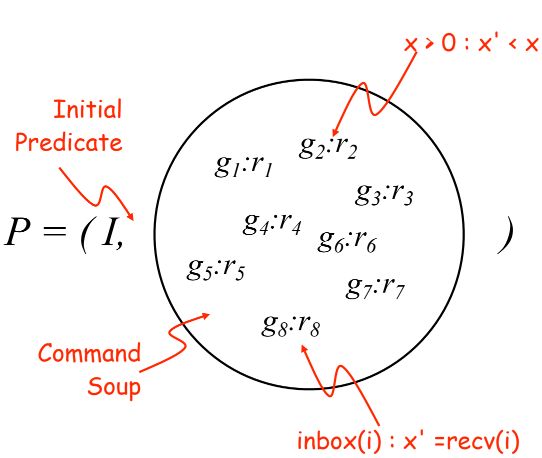
<u>Formal Results</u>

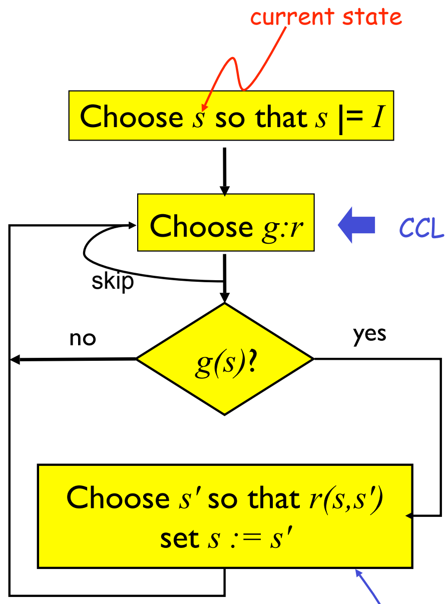
Formal semantics in transition systems and temporal logic. *RoboFlag* drill formalized and basic algorithms verified.

<u>Automated Verification</u>

CCL encoded in the *Isabelle* theorem prover; basic specs verified semi-automatically. Investigating various model checking tools.

Guarded Command Programs

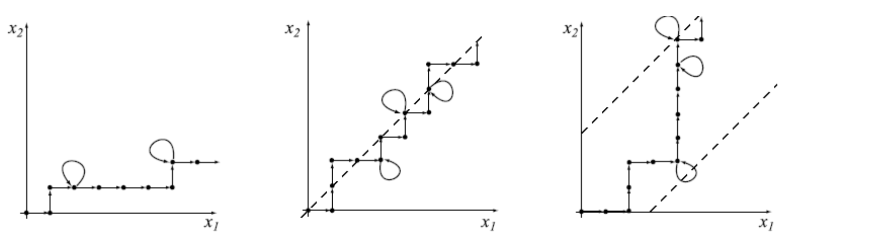




- Non-deterministic execution schedule models concurrency
- Easy to reason about programs
- Guarded commands = update functions

Any sequence of states produced by this process is a possible behavior of the system. We want to reason about them all.

Scheduling and Composition



UNITY
Each command must be executed infinitely often.

EPOCH
Each command is executed before any are again.

SYNCH(τ)
In any interval, the difference in the number of times any two commands are executed is $\leq \tau$.

$P(i)$	
Initial	$x_i = 0$
Commands	$true \ : \ x_i' = x_i + 1$

$Q = P(1) + P(2)$

Program composition:
$(I_1,C_1) + (I_2,C_2) = (I_1 \wedge I_2, C_1 \cup C_2)$

Thm: $SYNCH(1) \subseteq EPOCH \subseteq SYNCH(2) \subseteq SYNCH(3) \subseteq \ldots \subseteq UNITY$.

An Example CCL Program

```
include standard.ccl

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

program control() := {
  y := 0.0;
  u := 0.0;
  true : { u := -y };
};

program sys ( a, b, x0 ) := plant ( a, b, x0, 0.1 ) +
                            control ( 2*a/b ) sharing u, y;

exec sys ( 3.1, 0.75, 15.23 );
```

```
x = 3.216250
x = 3.095641
x = 2.979554
x = 2.867821
x = 2.760278
x = 2.656767
x = 2.557138
x = 2.461246
x = 2.368949
x = 2.280113
x = 2.194609
x = 2.112311
x = 2.033100
x = 1.956858
x = 1.883476
x = 1.812846
x = 1.744864
x = 1.679432
x = 1.616453
...
```

Structure of CCL Programs

```
program prog1 = {

   declarations

   initial {
     assignments
   }

   guard : { rules }
   guard : { rules }
   ...
};
```

- `program prog1 = {` — Declares a new program with name "prog1"
- *declarations* — Declare variables and functions to be used.
- `initial { assignments }` — Initialize state (variables and environment)
- *guard : { rules }* — Any number of "clauses". Guards are boolean expressions and rules are assignments to variables or control commands.

```
program prog3 ( ... ) :=
  prog1 ( ... ) +
  prog2 ( ... ) sharing x, y, z, ...;
```

This makes a new program with conjoined initial section and includes all clauses from prog1 and prog2. x, y and z are shared, other vars are local.

```
n {
  agent 0 gets prog0;
  agent 1 gets prog1;
  ...
}
```

For the simulator: assign programs to agents

```
exec prog ( 1.1, 2.0 );
```

Starts the interpreter.

CCL Language Features (optional)

Variables

- Can be of type constant, number or array

External functions

- Can be of type function, arrayfunction, boolean, with numerical arguments
- Can link to C/C++ functions
- `whoami, time, posx, posy, print, rand, reset, send_mesg, clear_box, sin, cos, abs, pos, vel, get_mesg, check_box,...`

Expressions

- Numeric (`1 + sin(x+y)/time()`) or boolean (`y[2] < y[3] || false`)

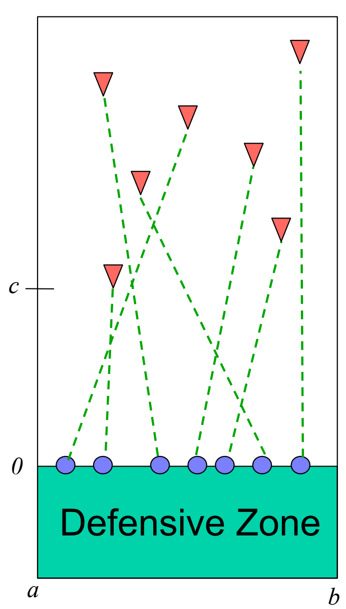
Communications

- Mailboxes: `send_mesg(to, arg`$_1$`, ..., arg`$_n$`)`, `recv_mesg (from)`, `check_box (from)`

Predefined Controllers

- Specified with the controller keyword
- `velcontrol, pd, force, pd_vehicle,...`

Example: RoboFlag Drill



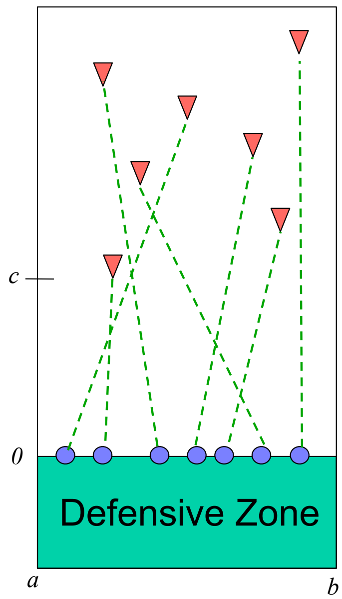
$Red(i)$	
Initial	$x_i \in [a,b] \wedge y_i > c$
Commands	$y_i > \delta \ : \ y_i' = y_i - \delta$
	$y_i \leq \delta \ : \ x_i' \in [a,b] \wedge y_i > c$

$$P_{Red}(n) = +_{i=1}^{n} Red(i)$$

$Blue(i)$	
Initial	$z_i \in [a,b] \wedge z_i < z_{i+1}$
Commands	$z_i < x_{\alpha(i)} \wedge z_i < z_{i+1} - \delta \ : \ z_i' = z_i + \delta$
	$z_i > x_{\alpha(i)} \wedge z_i > z_{i-1} + \delta \ : \ z_i' = z_i - \delta$

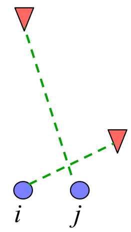
$$P_{Blue}(n) = +_{i=1}^{n} Blue(i)$$

RoboFlag Control Protocol



$\alpha(j)$ is too far down for i to get

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



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

$Proto(i)$	
Initial	$i \neq j \Rightarrow \alpha(i) \neq \alpha(j)$
Commands	$switch(i, i+1) \ : \ \alpha(i)' = \alpha(i+1)$ $\alpha(i+1)' = \alpha(i)$

$$P_{Proto}(n) = +_{i=1}^{n-1} Proto(i)$$

CCL Program for Switching Assignments

```
program Blue ( i ) := {

  red[alpha[i]][0] > blue[i] & blue[i] +
delta < toplimit i : {
    blue[i] := blue[i] + delta
  }

  red[alpha[i]][0] < blue[i] & blue[i] -
delta > botlimit i : {
    blue[i] := blue[i] - delta
  }

};
```

```
program Red ( i ) := {

  red[i][1] > delta : {
    red[i][1] := red[i][1] - delta
  }

  red[i][1] < delta : {
    red[i] := { rrand 0 n, rrand lowerlimit
n }
  }

};
```

```
fun r i j .
  if red[alpha[j]][1] < abs ( blue[i] -
red[alpha[j]][0] )
    then 1
    else 0
  end;

fun switch i j .
  r i j + r j i < r i i + r j j
  | ( r i j + r j i = r i i + r j j
    & red[alpha[i]][0]  > red[alpha[j][0] );

program ProtoPair ( i, j ) := {

  temp := 0;

  switch i j : {
    temp := alpha[i],
    alpha[i] := alpha[j],
    alpha[j] := temp,
  }

};
```

CCL/Temporal Logic Notation

Temporal logic

- ☐p — always p (invariance)
- ◊p — eventually p (guarantee)
- p → ◊q — p implies eventually q (response)
- p → q U r — p implies q until r (precedence)
- ☐◊p — always eventually p (progress)
- ◊☐p — eventually always p (stability)
- ◊p → ◊q — eventually p implies eventually q (correlation)
- ¬p — negation (not p)
- σ⟦F⟧ — true if a behavior σ satisfies a formula F

- P ⊨ F — ∀σ . σ⟦P⟧ ⇒ σ⟦F⟧ — P models F (any behavior consistent with a program satisfies a specified formula)

CCL

- skip — true : ∀v . v' = v — guarded command that does nothing
- p ↦ q — ☐(p ⇒ ◊q) — "p leads to q": if p is true, q will eventually be true
- p **co** q — ☐(p ⇒ [(q' ∨ skip]) ∧ ◊q']) — if p is true, then next time state changes, q will be true

Properties for RoboFlag program

Safety (Defenders do not collide)

$$z_i < z_{i+1} \ \mathbf{co}\ z_i < z_{i+1}$$

Stability (switch predicate stays false)

$$\forall i \ . \ y_i > 2\delta \wedge z_i + 2\delta < z_{i+1} \wedge \neg switch_{i,i+1} \ \mathbf{co}\ \neg switch_{i,i+1}$$

Robots are "far enough" apart.

"Lyapunov" stability

- Let ρ be the number of blue robots that are too far away to reach their red robots
- Let β be the total number of conflicts in the current assignment
- Define the Lyapunov function that captures "energy" of current state (V = 0 is desired)

$$V = \left[\binom{n}{2} + 1\right]\rho + \beta \qquad \rho = \sum_{i=1}^{n} r(i,i) \qquad \beta = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \gamma(i,j) \quad \text{where} \quad \gamma(i,j) = \begin{cases} 1 & \text{if } x_{\alpha(i)} > x_{\alpha(j)} \\ 0 & \text{otherwise} \end{cases}$$

- Can show that V always decreases whenever a switch occurs

$$\forall i \ . \ z_i + 2\delta m < z_{i+1} \wedge \exists j \ . \ switch_{j,j+1} \wedge V = m \ \ \mathbf{co}\ V < m$$

Sketch of Proof for RoboFlag Drill

More notation:

- Meaning of an action: *s* [[*a*]] *t* ≡ *a*(∀*v* : *s*[[*v*]] / *v*, *t*[[*v*]] / *v*')
  - Updates the state of the system by replacing all unprimed variables in *a* by their values under the state *s* and replacing all primed variables in *a* by their values under *t*
- Hoare triple notation: {*P*} *a* {*Q*} ≡ ∀ *s*, *t* . *s*[[P]] ^ *s* [[*a*]] *t* => *t*[[Q]]
  - True if the predicate *P* being true implies that Q is true after action *a*

Lemma (Klavins, 5.2) Let *P* = (*I, C*) be a program and *p* and *q* be predictates. If for all commands *c* in *C* we have {*p*} *c* {*q*} then *P* ⊨ *p* **co** *q*.

- If *p* is true then any action in the program *P* that can be applied in the current state leaves *q* true

Thm Prf(n) ⊨ ☐ $z_i < z_{i+1}$

- For the RoboFlag drill with *n* defenders and *n* attackers, the location of defender will always be to the left of defender *i*+1.

Proof. Using the lemma, it suffices to chech that for all commands *c* in *C* we have {*p*} *c* {*q*}. So, we need to show that if $z_i < z_{i+1}$ then any command that changes z_i or z_{i+1} leaves these unchanged. Two cases: i moves or i+1 moves. For the first case, {p} c {q} becomes

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

From the definition of the gaurded command, this is true. Similar for second case.

RoboFlag Simulation

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:

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

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

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

Initial:

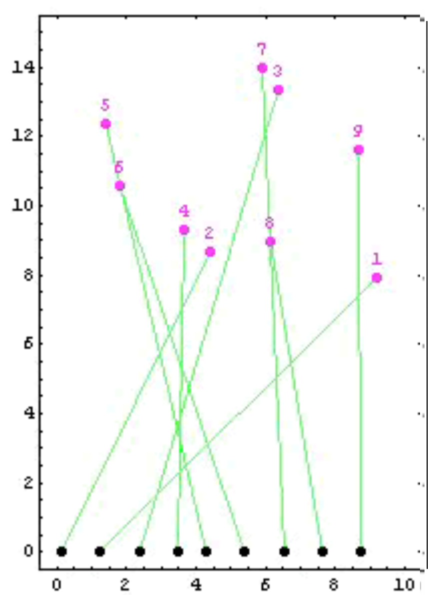
α is a bijection from $\{1, ..., n\}$ to $\{1, ..., n\}$.

Clauses:

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

...

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



Real-World Example: Lost Wingman Protocol (tomorrow)

Control T33 to follow F15 and to execute "lost wingman" during simulated communications loss.

sec.ccl

```
// F15 closed-loop dynamics and software
program F15() := F15_statemachine( 0 )                              // updates mode
               + F15_data_sender()               sharing u1, q1     // sends state to T33
               + F15_controller()                sharing u1, q1
               + F15_dynamics( {0.0, 0.0, 0.0, 30.0} ) sharing u1, q1;

// T33 closed-loop dynamics and software
program T33() := T33_statemachine( 0 )                              // updates mode
               + T33_data_receiver()      sharing u2, q2, q1_hat    // gets state from F15
               + T33_controller( 50, pi/6, {4.47, 0.32, 1.28 , 0.32 } ) sharing u2, q2, q1_hat
               + T33_dynamics( {-100.0, -40.0, 0.0, 30.0} ) sharing u2, q2;

program main() := F15()
                + T33() sharing q1, q2  // shares q1 just so draw_scene() works
                + draw_scene(400, 400) sharing q1, q2
                + sim_clock()
                + window_manager();
```

actual F-15 software

model of dynamics

Problem Framework

Agent dynamics

$$\dot{x}^i = f^i(x^i, u^i) \qquad x^i \in \mathbb{R}^n, u^i \in \mathbb{R}^m$$
$$y^i = h^i(x^i) \qquad y^i \in \mathrm{SE}(3),$$

Vehicle "role"

- $\alpha \in \mathcal{A}$ encodes internal state + relationship to current task
- Transition $\alpha' = r(x, \alpha)$

Communications graph $\mathcal{G}$

- Encodes the system information flow
- Neighbor set $\mathcal{N}^i(x, \alpha)$

Task

- Encode as finite horizon optimal control

$$J = \int_0^T L(x, \alpha, u)\, dt + V(x(T), \alpha(T)),$$

- Assume task is *coupled*

Strategy

- Control action for individual agents

$$u^i = \gamma(x, \alpha) \qquad \{g_j^i(x,\alpha) : r_j^i(x,\alpha)\}$$

$$\alpha^{i\,\prime} = \begin{cases} r_j^i(x,\alpha) & g(x,\alpha) = \text{true} \\ \text{unchanged} & \text{otherwise.} \end{cases}$$

Decentralized strategy

$$u^i(x,\alpha) = u^i(x^i, \alpha^i, x^{-i}, \alpha^{-i})$$
$$x^{-i} = \{x^{j_1}, \dots, x^{j_{m_i}}\}$$
$$j_k \in \mathcal{N}^i \qquad m_i = |\mathcal{N}^i|$$

- Similar structure for role update