Outline

• Review: networked control systems and cooperative control systems
• Asynchronous execution / group messaging systems (virtual synchrony)
• Verification of async control protocols for multi-agent, cooperative control
• Applications of model checking to Alice’s actuation interface
Networked Control Systems

(following P. R. Kumar)

- Actuation
- Process
- Sensing
- Online Model
- Inner Loop (PID, $H_\infty$)
- Mode and Fault Management
- Online Optimization (RHC, MILP)
- State Server (KF, MHE)
- Goal Mgmt (MDS)
- Attention & Awareness
- Memory and Learning

External Environment

Command:FIFO

Actuator State:Unreliable

Map:Causal

1-3 Gb/s

Online Optimization

10 Mb/s

Map:Causal

100 Kb/s

State:Unreliable

Feeder:Reliable
(Multi-) Vehicle Control Systems Framework

Agent dynamics - continuous
\[
\dot{x}^i = f^i(x^i, u^i) \quad x^i \in \mathbb{R}^n, u^i \in \mathbb{R}^m \\
y^i = h^i(x^i) \quad y^i \in \mathbb{R}^q
\]

Agent mode (or “role”) - discrete
- \( \alpha \in \mathcal{A} \) encodes internal state + relationship to current task
- Transition \( \alpha' = r(x, \alpha) \)

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

Communications channel
- Communicated information can be lost, delayed, reordered; rate constraints
\[
y^i_j[k] = \gamma y^i(t_k - t_j) \quad t_{k+1} - t_k > T_r
\]
- \( \gamma \) = binary random process (packet loss)

Task
- Encode task as finite horizon optimal control + temporal logic (assume coupled)
\[
J = \int_0^T L(x, \alpha, u) \, dt + V(x(T), \alpha(T)),
\]
\[
(\varphi_{\text{init}} \land \Box \varphi_e) \implies (\Box \varphi_s \land \Diamond \varphi_g)
\]

Strategy
- Control action for individual agents
\[
u^i = \gamma(x, \alpha) \quad \{ g^i_j(x, \alpha) : r^i_j(x, \alpha) \}
\]
\[
\alpha'^i = \begin{cases} 
  r^i_j(x, \alpha) & g(x, \alpha) = \text{true} \\
  \text{unchanged} & \text{otherwise.}
\end{cases}
\]

Decentralized strategy
\[
u^i(x, \alpha) = u^i(x^i, \alpha^i, y^{-i}, \alpha^{-i})
\]
\[
y^{-i} = \{ y^j_1, \ldots, y^j_{m_i} \}
\]
\[
j_k \in \mathcal{N}^i \quad m_i = |\mathcal{N}^i|
\]
- 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)
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

Questions
- How do we describe task in terms of LTL?
- Given a protocol, how do we prove specs?
- How do we design the protocol given specs?
\[ P(k_1, k_2) := \{ \]
\[ \text{initializers} \]
\[ \text{guard}_1 : \text{rule}_1 \]
\[ \text{guard}_2 : \text{rule}_2 \]
\[ \ldots \] \n\}

\[ S(k_1, k_2) := P(k_1, k_2) + C(k_1+1) \]

"soup" of guarded commands

composition = union

non-shared variables remain local to component programs

CCL: Computation and Control Language

Formal Language for Provably Correct Control Protocols

CCL Interpreter

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

Formal Results

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

Automated Verification

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

CCL Protocol for Decentralized Target Allocation
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

program composition:

\[(I_1, C_1) + (I_2, C_2) = (I_1 \land I_2, C_1 \cup C_2)\]

**UNITY**
Each command must be executed infinitely often.

**EPOCH**
Each command is executed before any are again.

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

\[
\begin{array}{c|c}
P(i) & \\
\hline
\text{Initial} & x_i = 0 \\
\text{Commands} & \text{true : } x'_i = x_i + 1 \\
\end{array}
\]

\[
Q = P(1) + P(2)
\]

Thm: \(SYNCH(1) \subseteq EPOCH \subseteq SYNCH(2) \subseteq SYNCH(3) \subseteq \ldots \subseteq UNITY\).
Example: RoboFlag Drill

\[
\begin{array}{l}
\text{Red}(i) \\
\text{Initial} & x_i \in [a, b] \land y_i > c \\
\text{Commands} & y_i > \delta : y'_i = y_i - \delta \\
& y_i \leq \delta : x'_i \in [a, b] \land y_i > c
\end{array}
\]

\[P_{\text{Red}}(n) = +\sum_{i=1}^{n} \text{Red}(i)\]

\[
\begin{array}{l}
\text{Blue}(i) \\
\text{Initial} & z_i \in [a, b] \land z_i < z_{i+1} \\
\text{Commands} & z_i < x_{\alpha(i)} \land z_i < z_{i+1} - \delta : z'_i = z_i + \delta \\
& z_i > x_{\alpha(i)} \land z_i > z_{i-1} + \delta : z'_i = z_i - \delta
\end{array}
\]

\[P_{\text{Blue}}(n) = +\sum_{i=1}^{n} \text{Blue}(i)\]
RoboFlag Control Protocol

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

\[ \text{switch}(i, j) = r(i, j) + r(j, i) < r(i, i) + r(j, j) \]
\[ \lor \quad (r(i, j) + r(j, i) = r(i, i) + r(j, j)) \land x_{\alpha(i)} > x_{\alpha(j)} \]

\[ \text{Proto}(i) \]

<table>
<thead>
<tr>
<th>\text{Initial Commands}</th>
<th>\text{Proto}(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i \neq j \Rightarrow \alpha(i) \neq \alpha(j) )</td>
<td>( i \neq j \Rightarrow \alpha(i) \neq \alpha(j) )</td>
</tr>
<tr>
<td>( \text{switch}(i, i + 1) : \alpha(i)' = \alpha(i + 1) )</td>
<td>( \text{switch}(i, i + 1) : \alpha(i)' = \alpha(i + 1) )</td>
</tr>
<tr>
<td>( \alpha(i + 1)' = \alpha(i) )</td>
<td>( \alpha(i + 1)' = \alpha(i) )</td>
</tr>
</tbody>
</table>

\[ P_{\text{Proto}}(n) = \sum_{i=1}^{n-1} \text{Proto}(i) \]
Properties for RoboFlag program

Safety (Defenders do not collide)
\[ z_i < z_{i+1} \quad \Box \quad z_i < z_{i+1} \]

Stability (switch predicate stays false)
\[ \forall i \cdot y_i > 2\delta \wedge z_i + 2\delta < z_{i+1} \wedge \neg switch_{i,i+1} \Box \neg switch_{i,i+1} \]

Robots are "far enough" apart.

“Lyapunov” stability

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

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

- Can show that \( V \) always decreases whenever a switch occurs

\[ \forall i \cdot z_i + 2\delta m < z_{i+1} \wedge \exists j \cdot switch_{j,j+1} \wedge V = m \quad \Box \quad V < m \]
Sketch of Proof for RoboFlag Drill

**Thm** $\text{Prf}(n) \models \Box z_i < z_{i+1}$

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

**More notation:**

- Hoare triple notation: $\{p\} a \{q\} \equiv \forall s \stackrel{a}{\rightarrow} t, s \models p \rightarrow t \models q$

- $\{p\} a \{q\}$ is 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 predicates. If for all commands $c$ in $C$ we have $\{p\} c \{q\}$ then $P \models p \text{ co } q$.

- If $p$ is true then any action in the program $P$ that can be applied in the current state leaves $q$ true
- Thus to check if $p \text{ co } q$ is true for a program, check each possible action

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

$$z_i < z_{i+1} \land (z_i < x_{\alpha(i)} \land z_i < z_{i+1} - \delta : z'_i = z_i + \delta) \implies z'_i < z'_{i+1}$$

From the definition of the guarded command, this is true. Similar for second case.
RoboFlag Simulation

**Exercise 1:** create a model of the RoboFlag drill in Promela and verify correctness using SPIN model checker

**Exercise 2:** create a specification for the RoboFlag drill and **synthesize** a (decentralized) protocol to solve it [later]
Planner Stack

Mission Planner performs high level decision-making
- Graph search for best routes; replan if routes are blocked

Traffic Planner handles rules of the road
- Control execution of path following & planning (multi-point turns)
- Encode traffic rules - when can we change lanes, proceed thru intersection, etc

Path Planner/Path Follower generate trajectories and track them
- Optimized trajectory generation + PID control (w/ anti-windup)
- Substantial control logic to handle failures, command interface, etc
Desired properties

- If *Estop Disable* is received, gcdrive state will be *Disabled* and acceleration will be ‘full brake’ forever
- *Estop Paused*: if not disabled, gcdrive will eventually enter *Paused* state and acceleration will be ‘full brake’ (not forever)
- *Estop Run*: if not *Disabled*, gcdrive will eventually be *Running* or *Resuming* (or receive another pause or disable command)
- If *Resuming*, eventually *Running* (or receive another pause or disable)
- If current mode is *Disabled, Paused, Resuming* or *Shifting*, full brake is commanded
- After receiving an *Estop Pause*, vehicle may resume operation 5 seconds after run is received (suffices to show that we transition from *Resuming* to *Running* via *Timeout*)
- ...
DGC Example: Changing Gear

Verify that we can’t drive while shifting or drive in the wrong gear

- Five component: follower Control, gcdrive Arbiter, gcdrive Control, actuators and network
- Construct temporal logic models for each component (including network)

Asynchronous operation

- Notation: Message_{mod,dir} - message to/from a module; Len = length of message queue
- Verify: follower has the right knowledge of the gear that we are currently in, or it commands a full brake.
  - ∅((Len(TransResp_{f,r}) = Len(Trans_{f,s})) ∧ TransResp_{f,r}[Len(TransResp_{f,r})] = COMPLETED ⇒ Trans_{f} = Trans)
  - ∅(Trans_{f} = Trans ∨ Acc_{f,s} = -1)
- Verify: at infinitely many instants, follower has the right knowledge of the gear that we are currently in, or we have hardware failure.
  - □◊(Trans_{f} = Trans = Trans_{f,s}[Len(Trans_{f,s})] ∨ HW failure)
Moving up the Planning Stack

How do we design control protocols that manage behavior

- Mixture of discrete and continuous decision making
- Insure proper response external events, with unknown timing
- Design input = specification + model (system + environment)
- Design output = finite state machine implementing logic

Approach: rapidly explore all trajectories satisfying specs

- Search through all possible actions and events, discarding executions that violate a set of (LTL) specifications
- Issue: state space explosion (especially due to environment)
- Good news: recent results in model checking for class of specs