Goals:
• Describe some types of specifications (contracts) for complex systems
• New concepts: program union and superposition, conditional properties
• Examples: mutex, revisited + RoboFlag drill

Reading:
• K. M. Chandy and J. Misra, Parallel Program Design: A Foundation, 1988 (Chapter 7)
• P. Sivilotti, Introduction to Distributed Algorithms, Chapter 8
Aircraft Vehicle Management Systems

How do we design software-controlled systems of systems to insure safe operation across all operating conditions (w/ failures)?
Design of Cyberphysical Systems (e.g. self-driving cars)

How do we manage the complexity?
- Abstraction
- A/G contracts
- Formal methods for verification/synthesis + model- & data-driven sims/testing

Layers of Abstraction
(most errors seem to occur here)
Structure of Specifications for a System

Partial module specifications
- Module variables
- Liveness goals
- Safety goals

Contract:
- Implementable
- Assume / guarantee
- In temporal logic

Assume/guarantee contracts
- Assume: properties of other components in the system
- Guarantee: properties that will hold for my component

\[ A_i \Rightarrow G_i \]
\[ G_2 \land G_3 \Rightarrow A_1, ~ G_1 \land G_3 \Rightarrow A_2, \ldots \]

“Horizontal” contracts
- A/G contracts within a layer

“Vertical” contracts
- A/G contracts between layers
Program Composition by Union

Multiple agents operating at same time
- Write separate programs for each agent and/or process
- Assume that operation is (almost) completely asynchronous (actually require sync(1) semantics…)
- Red robots = environment => get a type of “adversarial” interaction
  - Blue wants to block red
  - Red wants to evade blue

Program $P_{\text{red}}(i)$

| Initial | $x_i \in [\text{min}, \text{max}] \land y_i > \delta$ |
| Commands | $y_i - \delta > 0 : y'_i = y_i - \delta$ |

Program $P_{\text{blue}}(i)$

| Initial | $z_i \in [\text{min}, \text{max}] \land z_i < z_{i+1}$ |
| Commands | $z_i < x_{\alpha(i)} \land z_i < z_{i+1} - \delta : z'_i = z_i + \delta$ <br> $z_i > x_{\alpha(i)} \land z_i > z_{i-1} + \delta : z'_i = z_i - \delta$ |

Program $P_{\text{proto}}(i)$

| Initial | $\alpha(i) \neq \alpha(j)$ if $i \neq j$ |
| Commands | switch$(i, i + 1)$ : $\alpha(i)' = \alpha(i + 1)$ <br> $\alpha(i + 1)' = \alpha(i)$ |
Aside: The ‘Unless’ Property

P unless Q: if P is true, it remains true unless Q becomes true => Q acts like a program “gate”
- The only way for P to stop being true is if we “pass through” Q (and we require such an action exist)

Formal definition
- $P$ unless $Q \equiv (\forall a : a \in F : \{P \land \neg Q\} a \{P \lor Q\})$
- Alternative: $P$ unless $Q \equiv \{P \land \neg Q\} \text{next} \{P \lor Q\}$

If P holds at any point in the execution, then either
- Q never holds and P continues to hold forever
- Q holds eventually (and P holds until Q does)
- Also: if P holds at some point and then stops holding, Q must be true at that point

Example: different ways to say the value of x never decreases
- stable($x \geq k$)
- $x = k$ unless $x > k$
- $x \geq k$ unless $x > k$
- $x \geq k$ unless false

Some other properties and identities
- stable(P) $\equiv$ P unless false
- P ensures Q $\equiv$ P unless $Q \land \text{transient}(P \land \neg Q)$
- P unless $Q \land \text{invariant}(\neg Q) \implies$ stable(P)
Reasoning about Unions of Programs

Need to think about *combinations* of programs and how to proof things about them

- Write “property in F” if a given property holds in program F
- Write $H = F \parallel G$ for the “composition” $H$ of two “component” programs ($F$ and $G$)
- By default, share all variables with the same name (refine later)

Execution semantics

- To execute the union of a program, we just combine all of the rules into a single “bag”

Some properties of unions of programs

- $P$ unless $Q$ in $F \parallel G \equiv (P$ unless $Q$ in $F) \land (P$ unless $Q$ in $G)$
  - Why is this true? A: __________________________
- $P$ ensures $Q$ in $F \parallel G \equiv [P$ ensures $Q$ in $F \land P$ unless $Q$ in $G] \lor [P$ ensures $Q$ in $G \land P$ unless $Q$ in $F]$
  - Why is this not just $(P$ ensures $Q$ in $F) \land (P$ ensures $Q$ in $G)$?
  - A: __________________________________________
- $FP$ of $F \parallel G \equiv (FP$ of $F) \land (FP$ of $G)$
- $(P$ unless $Q$ in $F) \land (stable(P)$ in $G) \Rightarrow P$ unless $Q$ in $F \parallel G$
- Locality: $P$ is *local* to $F$ if it only uses variables in $F$. $local(P) \Rightarrow (P$ in $F \equiv P$ in $F \parallel G)$
Conditional Properties

Properties with hypothesis (assume) and conclusion (guarantee)
- For composite program $H = F \parallel G$, hypotheses & conclusions can be about $F$, $G$, or $H$
- Use conditional properties to prove properties without the entire program description

Example:

Program $F$

```plaintext
var $x, y : \text{integers}$
assign $(x \leq 0 \land y > 0) \rightarrow y := -y$
\[
\begin{array}{c}
\mid \\
\hline
x := -1
\end{array}
\]
```

- Let $G$ be any program that only shares the variable $y$. Show that the following conditional property is satisfied
  - Assume: $y \neq 0$ is stable in $F \parallel G$
  - Guarantee: $y > 0 \sim y < 0$ in $F \parallel G$

Proof

- Step 1: true $\sim x \leq 0$ in $F \parallel G$ \hspace{1cm} Why: ______________________________
- Step 2: $x \leq 0 \land y \neq 0 \sim y < 0$ in $F \parallel G$ \hspace{1cm} Why: ______________________________
- Now use PSP: $(P \sim Q) \land (R \text{ next } S) \Rightarrow (P \land R) \sim ((R \land Q) \lor (\neg R \land S))$
  - $P = \text{true}$
  - $Q = x \leq 0$ \hspace{1cm} $\Rightarrow y \neq 0 \sim (x \leq 0 \land y \neq 0) \sim y < 0$
  - $R = S = (y \neq 0)$
Superposition

Provide a mechanism for structuring a program as a set of “layers”

- Let G be a program that we wish to create by superposition from a program F
- Augmentation rule: An action \( a \) in the underlying program (F) may be transformed into an action \( a || b \) where \( b \) does not assign variables in F
- Restricted union rule: An action \( b \) may be added to F provided that \( b \) does not modify any of F’s variables

Theorem
Every property of the underlying program is a property of the transformed program

- Proof for augmentation: if \( \{P\} a \{Q\} \) holds then \( \{P\} a || b \{Q\} \) also holds
- Proof for restricted union: \( \text{local}(P) \Rightarrow (P \in F \equiv P \in F || G) \)

Example: detect whether a program has executed 10 actions (alternative: terminated)

Program \( detect10-aug \)

| initial | count = 0 || claim = false |
|---------|-----------------------------|
| transform | each statement \( s \) in F to | \( s || count := count + 1 \)
|         | \( \| claim := count \geq 10 \) |

Program \( detect10-augunion \)

| initial | count = 0 || claim = false |
|---------|-----------------------------|
| transform | each statement \( s \) in F to | \( s || count := count + 1 \)
|         | add | \( claim := count \geq 10 \) |
Example: Specification for Mutual Exclusion

UNITY style design specification format for transformed program $H = F' \parallel G$

- Specification of $F$: list of properties for $F$ + description of shared variables
  - Unconditional properties apply to $F$
  - Conditional properties apply to $H = F' \parallel G$

- Specification of $H$: list of (unconditional) properties that should be true for the composite program

- Constraints: Variables in $F$ that can be accessed from outside $F$

Example: mutual exclusion

- Properties for program user ($u = U_i$)
  - $u.\text{mode}=\text{NC}$ unless $u.\text{mode}=\text{TRY}$
  - stable($u.\text{mode}=\text{TRY}$)
  - $u.\text{mode}=\text{CS}$ unless $u.\text{mode}=\text{NC}$
  - Conditional property
    - $A$: $(\forall u,v : u \neq v : \neg (u.\text{mode} = \text{CS} \land v.\text{mode} = \text{CS}))$
    - $G$: $(\forall u : u.\text{mode} = \text{CS} \Rightarrow u.\text{mode} = \text{NC})$

- Properties for program $\text{mutex}$ ($H$)
  - $u.\text{mode} = \text{TRY} \Rightarrow u.\text{mode} = \text{CS}$
  - invariant($\neg (u.\text{mode} = \text{CS} \land v.\text{mode} = \text{CS} \land u \neq v)$)

- Constraints: what mutex protocol can access
  - Only non-local variable is the $u.\text{mode}$
  - $(\forall u : \text{stable}(u.m=\text{CS}))$ in $G$
  - $(\forall u : \text{stable}(u.m=\text{NC}))$ in $G$
Summary: Specifications and Composition

Key ideas:

- Decompose programs (and specifications) into interconnected modules
- Union = combining two or more programs with a shared set of variables
- Superposition = adding new rules to a program

Conditional properties = A/G contracts

- Use hypothesis/conclusion (assume/guarantee) structure to reason about properties of composite programs
- Can often reason about program composition without looking at specific actions (just specs on those actions)

Wednesday: refining specifications

- How do we simplify a complex specification and come up with a satisfying program
- Example: dining philosophers