Caltech

CS 142: Lecture 7.1
Program Composition and Refinement

Richard M. Murray

13 November 2019

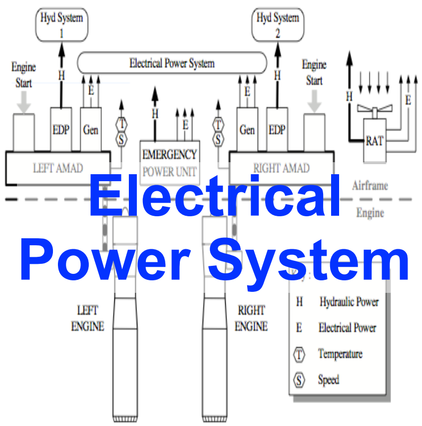
Goals:

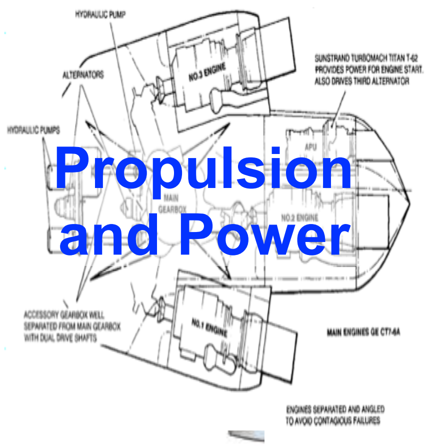
- Describe some types of specifications (contracts) for complex systems
- New concepts: program union and superposition, conditional properties
- Describe how to *refine* specifications for complex problems
- Examples: mutex and dining philosophers, revisited

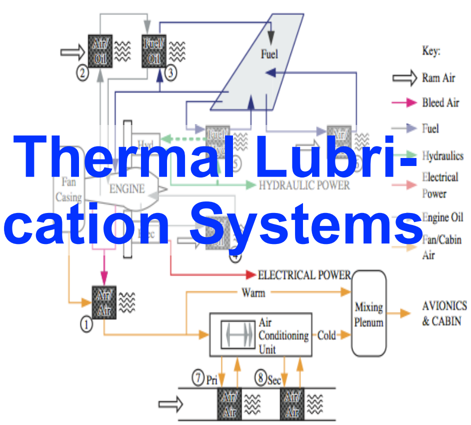
Reading:

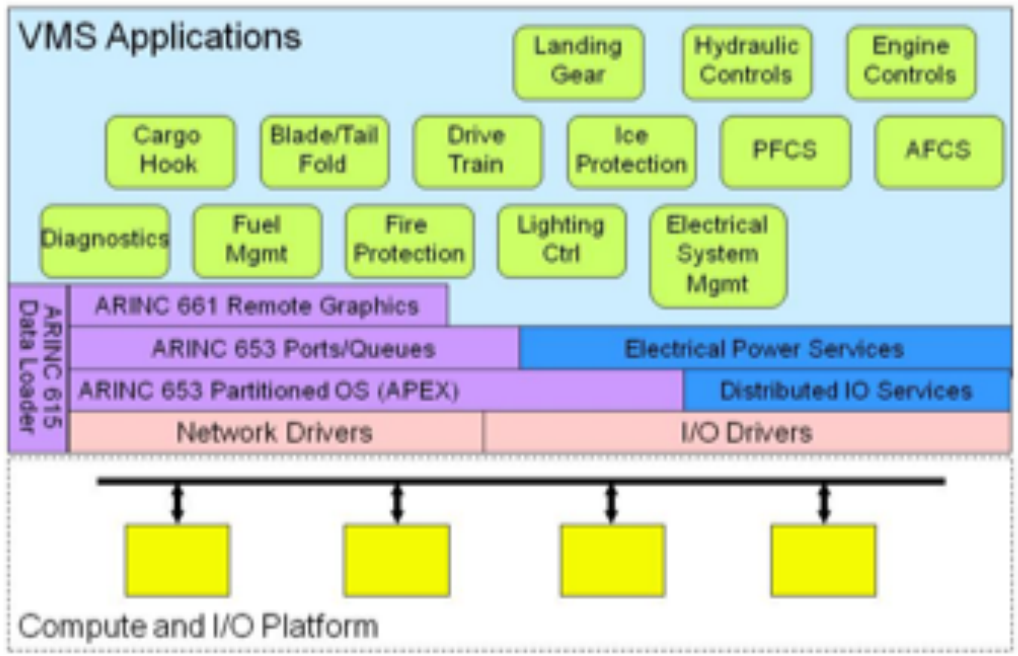
- K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, 1988 (Chapter 7) [posted on Moodle]
- K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, 1988 (Chapter 12) [posted on Moodle]
- P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 8

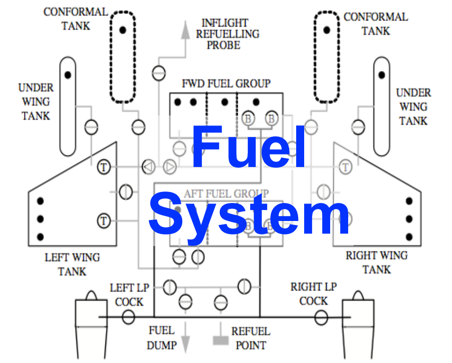
Aircraft Vehicle Management Systems

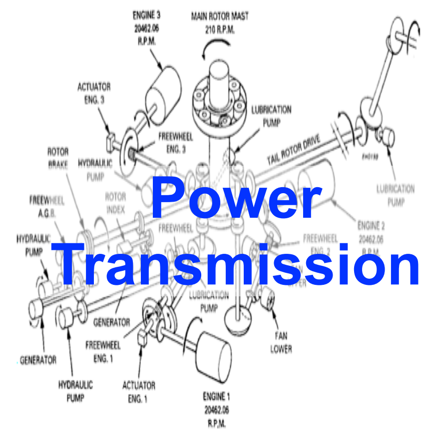




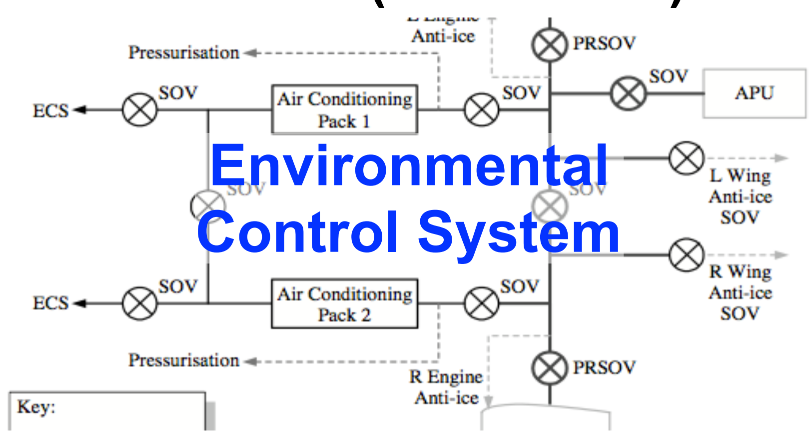


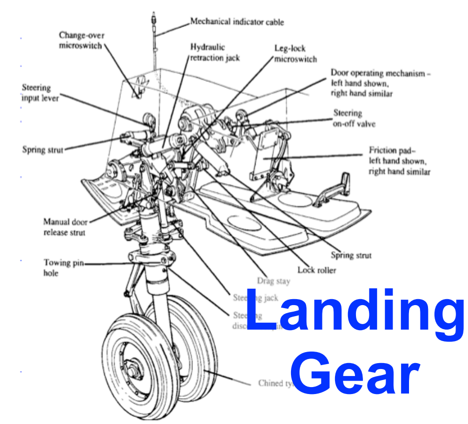




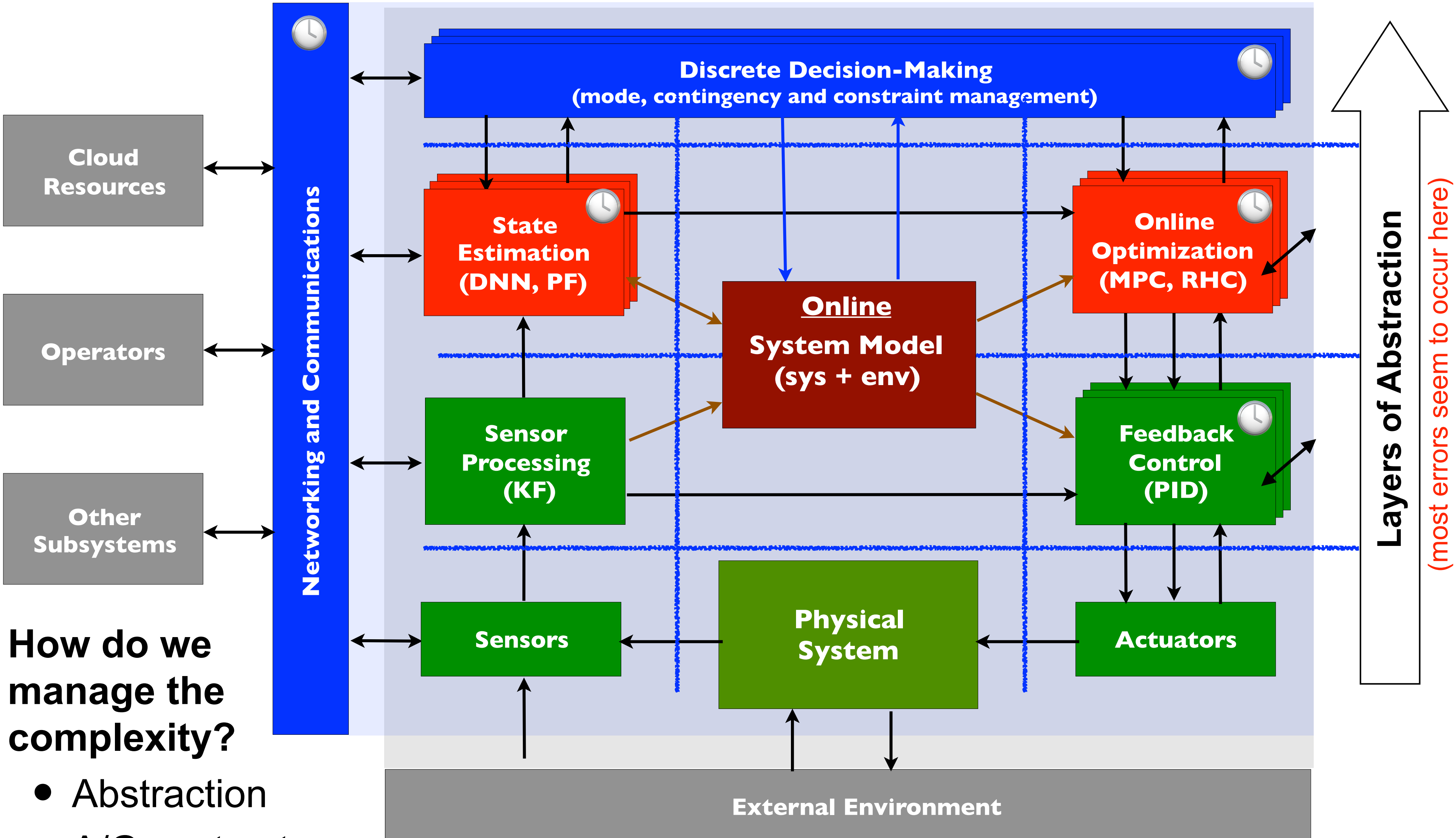


How do we design software-controlled systems of systems to insure safe operation across all operating conditions (w/ failures)?





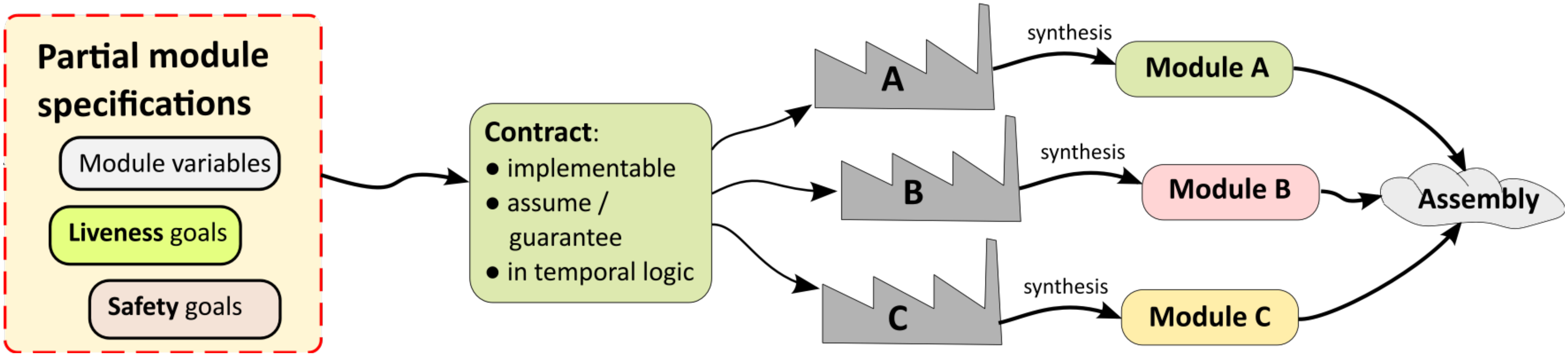
Design of Cyberphysical Systems (e.g. self-driving cars)



How do we manage the complexity?

- Abstraction
- A/G contracts
- Formal methods for verification/synthesis + model- & data-driven sims/testing

Structure of Specifications for a System



Assume/guarantee contracts

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

$$A_i \Rightarrow G_i$$

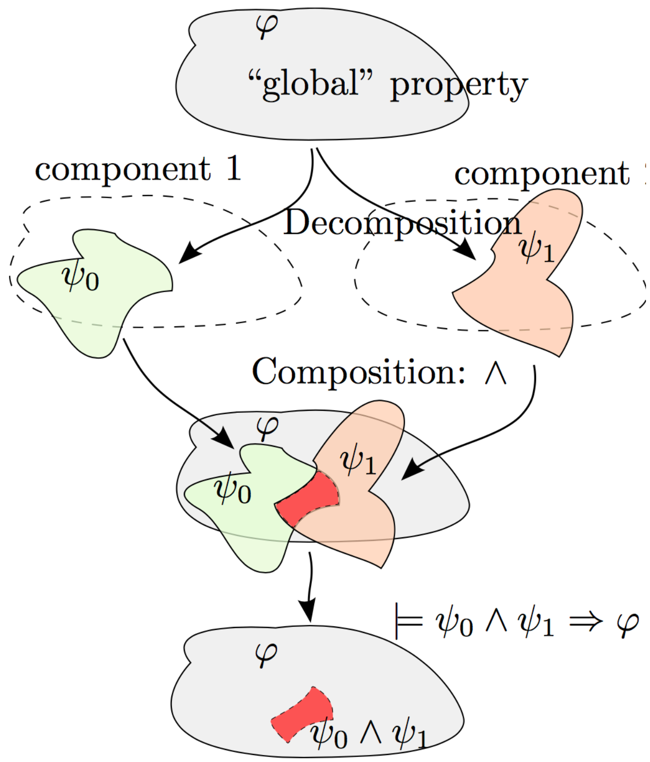
$G_2 \wedge G_3 \Rightarrow A_1,\ G_1 \wedge G_3 \Rightarrow A_2,\ \ldots$

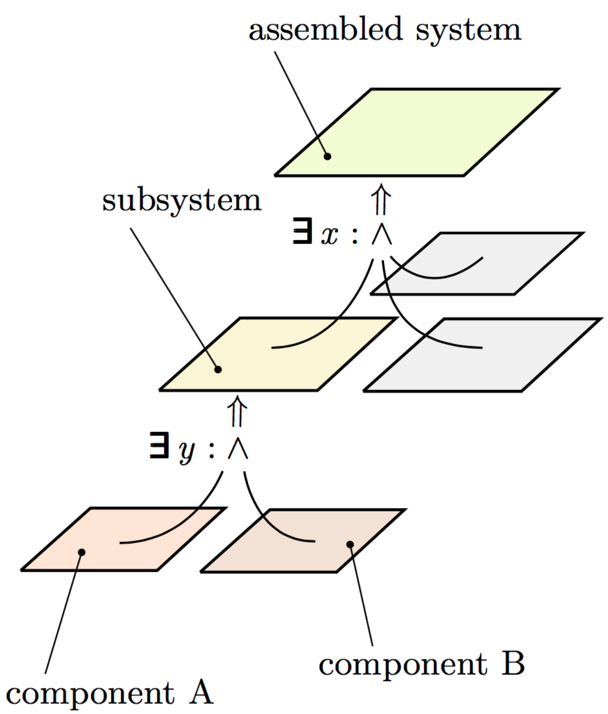
"Horizontal" contracts

- A/G contracts within a layer

"Vertical" contracts

- A/G contracts between layers

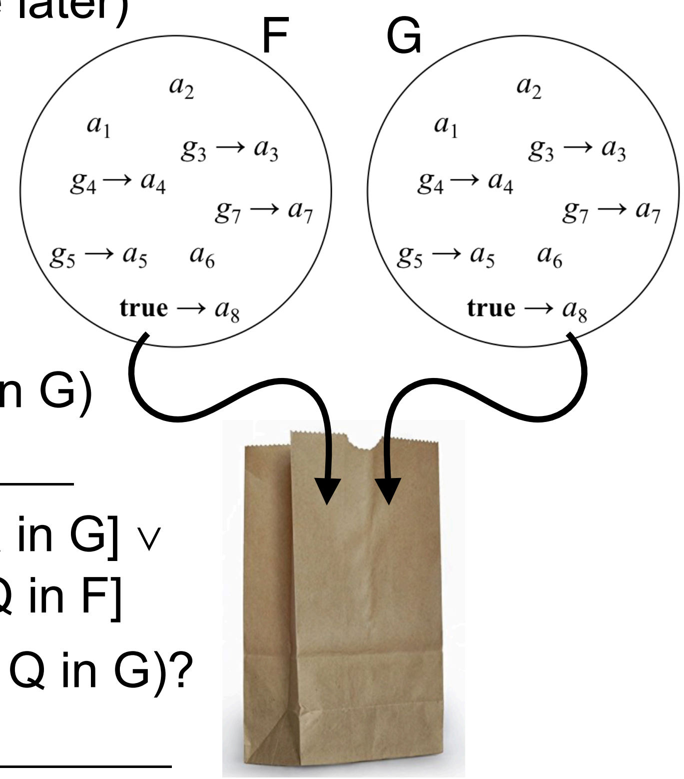




Reasoning about Unions of Programs

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 (also F ⊧ P)
- Write H = F ‖ 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 ‖ G ≡ (P **unless** Q in F) ∧ (P **unless** Q in G)
  - Why is this true? A: ______________________________
- P **ensures** Q in F ‖ G ≡ [P **ensures** Q in F ∧ P **unless** Q in G] ∨ [P **ensures** Q in G ∧ P **unless** Q in F]
  - Why is this not just (P **ensures** Q in F) ∧ (P **ensures** Q in G)?
  - A: ____
- FP of F | G ≡ (FP of F) ∧ (FP of G)
- (P **unless** Q in F) ∧ (**stable**(P) in G) ⇒ P **unless** Q in F | G
- Locality: P is *local* to F if it only uses variables in F. **local**(P) ⇒ (P in F ≡ P in F ‖ G)

Conditional Properties

Properties with hypothesis (assume) and conclusion (guarantee)

- For composite program H = F $[\!]$ G, hypotheses & conclusions can be about F, G, or H
- Use conditional properties to prove properties without the entire program description

Example:

$$\begin{array}{ll} \textbf{Program} & F \\ \textbf{var} & x, y : \text{integers} \\ \textbf{assign} & \\ & (x \leq 0 \wedge y > 0) \rightarrow y := -y \\ [\!] & x := -1 \end{array}$$

- 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 $[\!]$ G
  - Guarantee: $y > 0 \rightsquigarrow y < 0$ in F $[\!]$ G

Proof

- Step 1: true $\rightsquigarrow x \leq 0$ in F $[\!]$ G   Why: ____________________________
- Step 2: $x \leq 0 \wedge y \neq 0 \rightsquigarrow y < 0$ in F $[\!]$ G   Why: ____________________________
- Now use PSP: $(P \rightsquigarrow Q) \wedge (R \ \textbf{next} \ S) \Rightarrow (P \wedge R) \rightsquigarrow ((R \wedge Q) \vee (\neg R \wedge S))$
  - P = true
  - $Q = x \leq 0$ $\Rightarrow y \neq 0 \rightsquigarrow (x \leq 0 \wedge y \neq 0) \rightsquigarrow y < 0$
  - $R = S = (y \neq 0)$

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: **local**(P) ⇒ (P in F ≡ P in F ⫾ G)

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

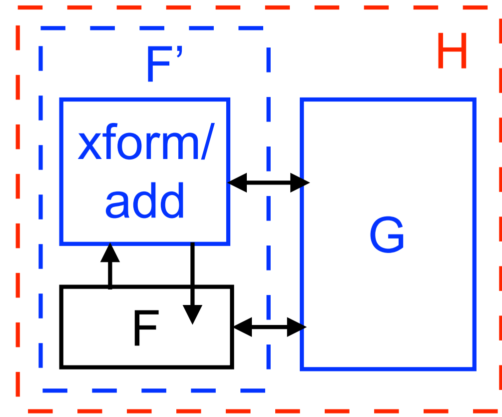
$$
\begin{array}{ll}
\textbf{Program} & \textit{detect10-aug} \\
\textbf{initial} & count = 0 \parallel claim = \textbf{false} \\
\textbf{transform} & \\
\quad \text{each statement } s \text{ in F to} & \\
\quad s \parallel count := count + 1 & \\
\quad\ \ \parallel claim := count \geq 10 &
\end{array}
$$

$$
\begin{array}{ll}
\textbf{Program} & \textit{detect10-augunion} \\
\textbf{initial} & count = 0 \parallel claim = \textbf{false} \\
\textbf{transform} & \\
\quad \text{each statement } s \text{ in F to} & \\
\quad s \parallel count := count + 1 & \\
\textbf{add} & \\
\quad claim := count \geq 10 &
\end{array}
$$

Example: Specification for Mutual Exclusion

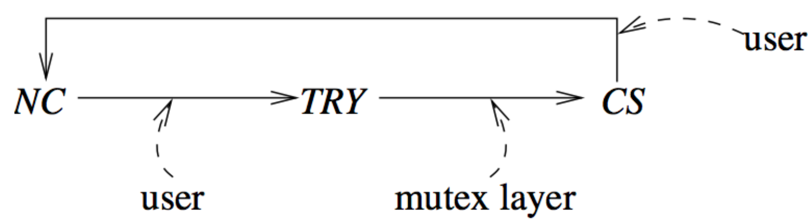
UNITY style design specification format for transformed program H = F’ ⫿ G

- Specification of F: list of properties for F + description of shared variables
  - Unconditional properties apply to F
  - Conditional properties apply to H = F’ ⫿ 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

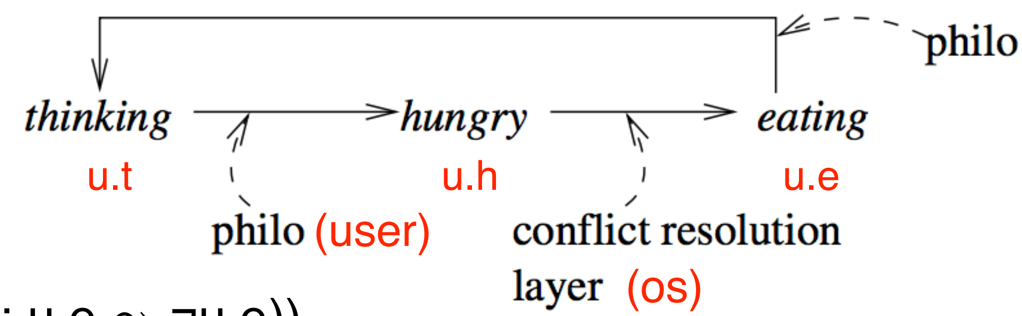


Example: mutual exclusion

- Properties for program user ($u = U_i$)
  - u.mode=NC **unless** u.mode=TRY
  - **stable**(u.mode=TRY)
  - u.mode=CS **unless** u.mode=NC
  - Conditional property
    - A: $(\forall u,v : u \neq v : \neg(u.mode = CS \wedge v.mode = CS))$
    - G: $(\forall u :: u.mode = CS \leadsto u.mode = NC)$
- Constraints: what mutex protocol can access
  - Only non-local variable is u.mode
  - $(\forall u : stable(u.m=CS))$ in G
  - $(\forall u : stable(u.m=NC)$ in G
- Properties for program *mutex* (H)
  - $u.mode = TRY \leadsto u.mode = CS$
  - $invariant(\neg(u.mode = CS \wedge v.mode = CS \wedge u \neq v\ ))$



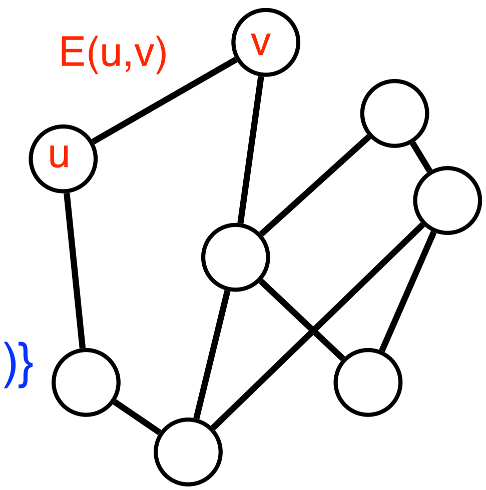
Program Specification (Dining Philosophers)



User process specification

- udn1: u.t **unless** u.h in user
- udn2: **stable**(u.h) in user
- udn3: u.e **unless** u.t in user
- udn4: $(\forall u,v : E(u,v) : \neg(u.e \wedge v.e)) \Rightarrow (\forall u :: u.e \leadsto \neg u.e))$

Specification of composite program



- dn1: (safety): **invariant** $(\neg(u.e \wedge v.e \wedge E(u,v))$ in user | os
- dn2: (progress): $u.h \leadsto u.e$ in user | os

Constraints on conflict resolution layer (os)

- odn1: **constant**(u.t) in os {constant(P) = stable(P) ^ stable(!P)}
- odn2: **stable**(u.e) in os
- Derived properties of os
  - **stable**(¬u.h) in os
  - u.h **unless** u.e in os

CM88 key:

dn = dining (philosophers)
udn = user process spec
odn = os process spec

Given these specs, how do we proceed?

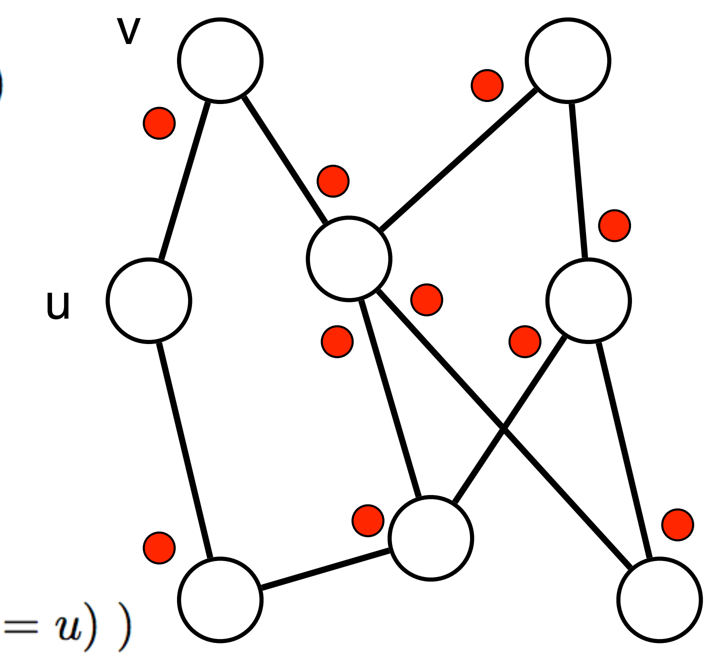
- Need to define a "program" that implements the "os" function in a distributed fashion
- OK to assume listed properties about agents
- Approach: write *specs* for os, then write code

Specification Refinement #1: Safety

Original specification of composite program:

dn1: $(\forall u, v :: \textbf{invariant}.(\neg(E(u,v) \land u.e \land v.e)))$

- Can implement this invariant by making use of a token (*a la* mutual exclusion)
- For each edge (u,v) in the graph, establish a token fork(u, v) that keeps track of who has access to the shared resource (fork) at the current time
- New spec: if u is eating (in CS), then it must have the token



odn9: $(\forall u, v :: \textbf{invariant}.(u.e \land E(u,v) \Rightarrow fork(u,v) = u))$

- New spec satisfies the old spec since token can only be in one place at a time

Implement that idea of a token by *refining* the specification

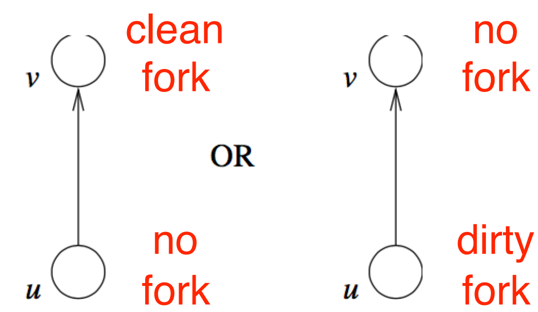
- Add new variables/functions and write specification in term of those quantities
- New specification should satisfy the original specification
- In setting up the new specification, you are making a choice about program structure
  - For dining philosophers, this refinement means we will use a token-based approach to enforce mutual exclusion on each edge

Additional Refinements: Priority, Token Request

Need to break the symmetry between philosophers

- Basic idea: establish some sort of priority on the graph

$$u < v \quad \equiv \quad (fork(u,v) = v \wedge clean(u,v)) \vee (fork(u,v) = u \wedge \neg clean(u,v))$$



Establish desired properties (informal refinement)

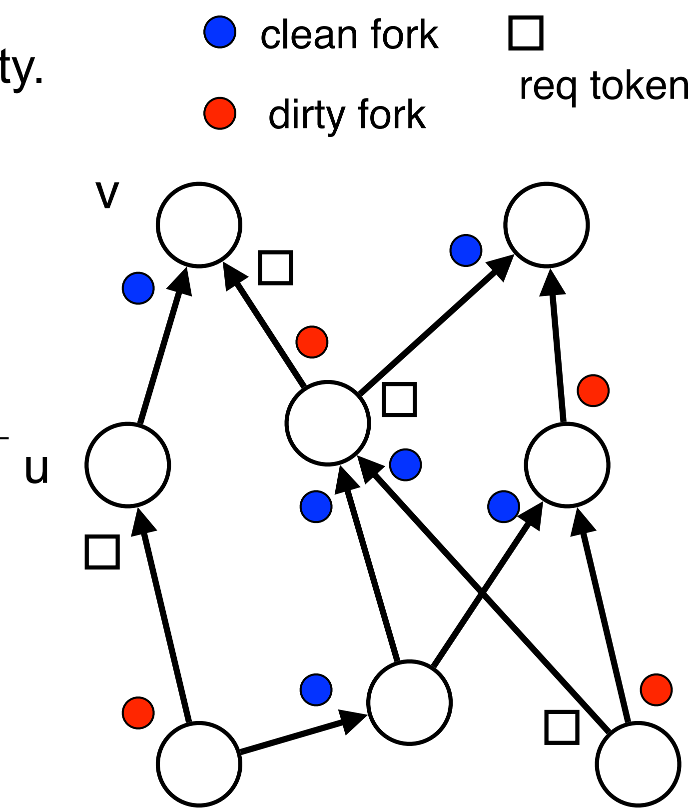
1. An eating process holds all its forks and the forks are dirty.
2. A process holding a clean fork continues to hold it (and it remains clean) until the process eats.
3. A dirty fork remains dirty until it is sent from one process to another (at which point it is cleaned)
4. Clean forks are held only by hungry philosophers

Problem: how do we know if our neighbor is hungry?

- Need this in order to implement previous spec

Solution: add a “request token” req(u,v) to each edge

- Idea: if agent is hungry, doesn’t have fork, and has the request token, then send request to v (set req(u,v) = v)

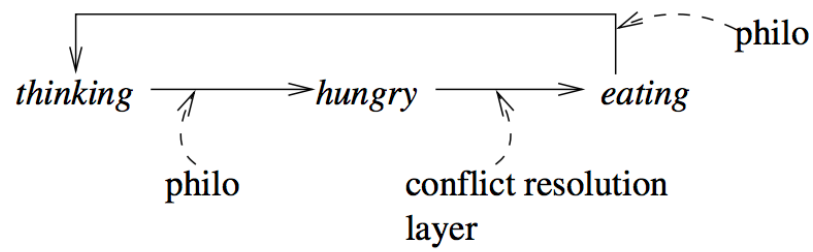


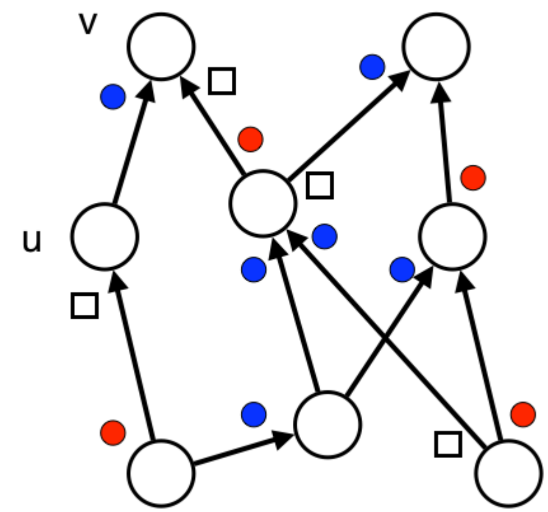
Approach: refine specifications and use this to define the program (for the os)

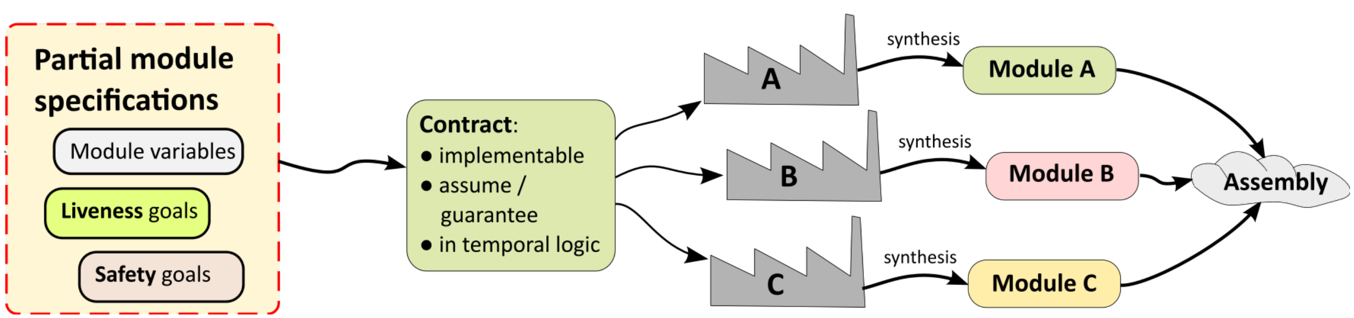
Summary: Composition and Refinement

Key ideas:

- Specifications for composed systems
  - Properties of the underlying process (user)
  - Properties of the composed system (user | os)
  - Constraints on access to user processes
- Design via successive refinement
  - Refine properties to establish program structure
  - Each refinement solves problem from previous level (and satisfies the prior specs)
  - Final specification can be converted to code
- Advantages of this approach
  - Maintain a formal proof structure throughout
  - Painful, but necessary for safety critical systems!







Wed: global snapshots

Next week: fault tolerance

- Byzantine agreement
- Paxos algorithm