Motivation: Compositionality

- Compositionality $\equiv$ Modularity
  - Ability to “compose” verifications of modules to verify a larger system
  - Logic example: Verify a program using pre- and post-conditions of verified procedures
  - Practical requirement: Verification or analysis results must be summarizations
- Compositionality in finite-state verification
  - Hierarchical analysis, summarizing results at each level
  - Potentially control state-space explosion
Non-Compositional Analysis

- We cannot find all behaviors of \( P|Q|R \) by finding behaviors of \( P|Q \) then composing with \( R \).

Adding Compositionality ...

- We want algebraic structure
  - Commutativity, associativity, and a congruence
    - e.g., \( A + B = C \Rightarrow A + D + B = A + B + D = (A + B) + D = C + D \)
  - Needed:
    - Account for “potential” behaviors of a subsystem
      - in \( P|Q|R \), the partial result \( P|Q \) should include action \( b \)
    - ... but limit to interface actions
      - record “potential” behaviors only if they are visible outside a module (e.g., actions \( a \) and \( b \) don’t matter to process \( R \))
    - ... and simplify subsystem analyses
      - the difference between \([a]\) and \([b]\) should not matter outside the subsystem \( P|Q\)
Processes as Terms

- Description of cooperating processes
  - Terms: similar to regular expressions
    - Context free processes are describable but too hairy
  - Process graphs: state machines denoted by terms
    - Regular processes denote finite-state process graphs
- Algebraic laws
  - Associative, commutative laws and substitution of equals for equals (and “less for equals”) for incremental reasoning:
    \[ X = A || B \text{ implies } X || C = A || B || C \] (equivalence)
    \[ X \leq A || B \text{ implies } X || C \leq A || B || C \] (preorder)

Process Expressions

- Constants
  - \( \delta \) (deadlock, or no action)
  - \( \tau \) (internal, unobservable action, similar to \( \varepsilon \))
  - \( a, b, c, \ldots \) Observable actions
- Expressions formed from
  - ; (sequence, with \( a;b \) abbreviated as \( ab \))
  - + (choice)
  - | (synchronization of 2 events)
    \[ aP || bQ = (a||b)(P || Q) + a(P || bQ) + b(aP || Q) \]
Why $\tau \neq \varepsilon$

The other axioms of regular expressions come across without change, but note $ab + ac = a(\tau b + \tau c)$.

Synchronization

- $aP||bQ = (a|b)(P||Q) + a(P||bQ) + b(aP||Q)$
  - i.e., one moves first or else they move together
- In general, $a|b$ is some action $c$
- In CCS, $a|-a$ is $\tau$, other pairs are $\delta$
  - synchronization is rendezvous between action and co-action, and rendezvous is unobservable by other processes
- In CSP, $a|a$ is $a$, other pairs are $\delta$
  - synchronization is agreement to do the same thing
Product of Processes

This is progress?
(not unless we can simplify the intermediate product)

Equivalence and Congruence

- Language equivalence is too coarse:
  - $ab + ac = a(b+c)$, which we have seen is wrong
  - We want something nearly as coarse, but preserving deadlock, cheap to check and compute quotients

- Bisimulation:
  - $P=Q$ iff $P \xrightarrow{-a} P'$ implies $Q \xrightarrow{-a} Q'$ and $P'=Q'$
  - $Q \xrightarrow{-a} Q'$ implies $P \xrightarrow{-a} P'$ and $P'=Q'$
  - Strong bisim equivalent if we consider $t$ an action
  - Weak bisim equivalent if an action is $at^*$
  - Cheap to compute: similar to DFA minimization
Abstraction and Restriction

- Abstraction: Substitute $\tau$ for $a$
  - Meaning: I don’t care about $a$ in this context
  - Especially: I don’t interact with that action
- Restriction: Substitute $\delta$ for $a$
  - Meaning: That can’t happen in this context
  - Especially: That interface isn’t visible here
- At module boundaries,
  - Abstract actions that can happen “in the box”
  - Restrict actions in internal interfaces

Simplifying $P \parallel Q$

- Restrict $a, b$ and abstract $[a], [b]$

\[
P \parallel Q : \quad \begin{array}{c}
P \parallel Q \\\\{a, b\} : \quad \end{array}
\]

\[
(P \parallel Q) \backslash \{a, b\} : \quad \begin{array}{c}
\tau \\
\end{array}
\]

\[
\begin{array}{c}
\tau \\
\end{array}
\]

\[
\begin{array}{c}
c \\
\end{array}
\]
Preorder and Precongruence

- We don't always want equivalence
  - We want to permit looser specs, like a super/sub-type relation among processes
  - Example: Bounded queue of unspecified length
  - A “preorder” relates specification ≤ implementation
- The “testing” preorders
  - may: language inclusion
    - if p may pass a test, q may pass that test
  - must: failures inclusion
    - if p must pass a test, q must pass that test

Why should I care?

- Congruence (or preferably pre-congruence) is a useful definition of conformance of an implementation to an interface specification
- Process product permits one to say “these processes together meet that spec”
- Abstraction and restriction are the semantic building blocks for modularity
- Algebraic structure is essential (but not sufficient) for reasoning hierarchically about complex systems
State-space exploration example: 
Alternating Bit Protocol

Alternating Bit Protocol: 
After reduction

- After restriction and abstraction, process graphs can be reduced to equivalent form with respect to a congruence relation

... but radical reductions in process graph size occur only when the system to be analyzed is “well-structured”
Scalable analysis

- When compositional analysis “works”, reductions at intermediate steps keep state-space to manageable proportions.
- The question is, when does (or can) it work?

An example (redesigned)
Compositional analysis of revised design

Experience with Compositional Analysis using Process Algebra

- Has worked well for well-structured designs, poorly for code and “as built” designs
- (Re-)structuring for analysis is often necessary
  - Analyzable designs are more understandable and modifiable
  - BUT ... real designs are seldom structured as we want
  - AND WORSE ... there are good reasons for “bad” structure in source code
    - We must accept that the relation between a verified design and the “as built” structure of a system will not be simple