CIS 673: Lecture 11: Oct 11
Automata theoretic approach

- Use automata instead of temporal logic, to specify requirements

- Automata-theoretic verification:
  - Module is viewed as a generator of a language over strings of observations
  - Specification is an automaton that accepts good behaviors
  - Verification problem is to check if every string generated by module is accepted by specification

- Theory of formal languages provides conceptual framework

- Sample tool: COSPAN at Bell Labs (FormalCheck)

- For now, automata over finite strings

Safe Automaton Logic: SAL

- Syntax: Boolean combinations of specification automata

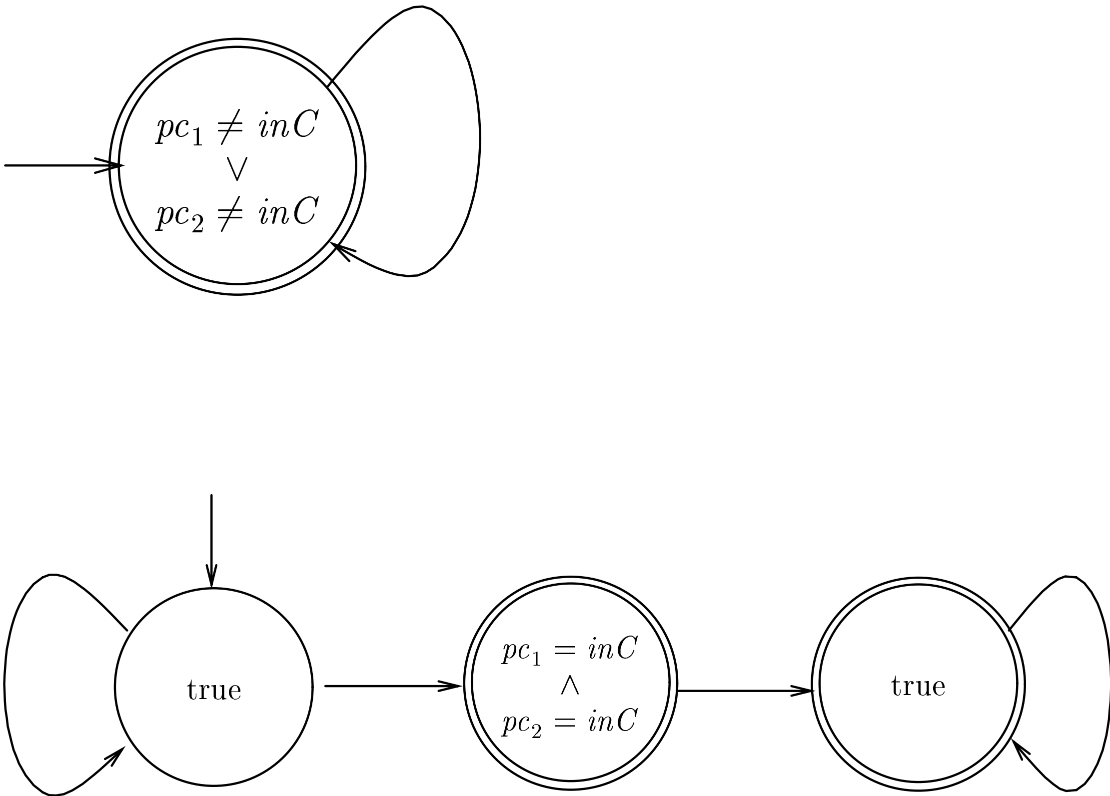
$$\phi \ ::= \ M \mid \neg\phi \mid \phi \vee \phi$$

- A formula of SAL can be interpreted at states of K if if each observation of K is a valuation for a superset of the variables appearing in the observations of all automata occurring in ϕ

- Semantics: A state s of K satisfies the SAL formula M if for every source-s trajectory $\overline{s}_{0..m}$ of K there is a trace $\overline{a}_{0..m} \in L_M$ such that for all i, $s_i \models a_i$

- Like other state logics, we can define characteristic regions, satisfaction $K \models \phi$ etc.

Specifying Mutual Exclusion



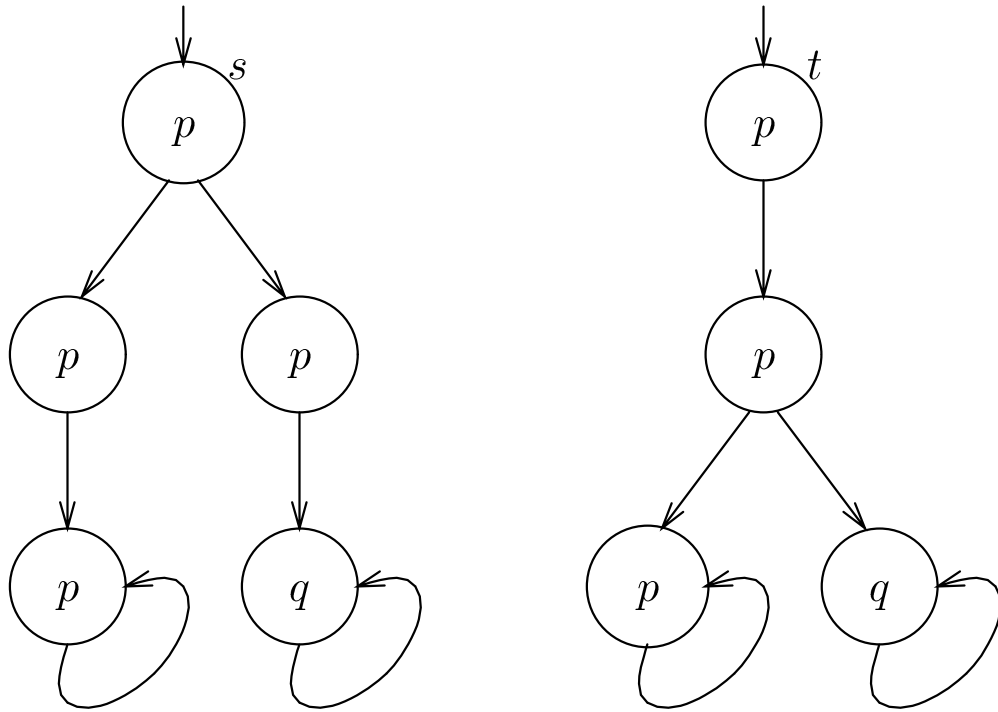
Trace Equivalence

- Two states s and t of an observation structure K are trace equivalent, denoted $s \simeq^L t$, if $L_K(s) = L_K(t)$.

- Trace equivalence is less distinguishing than bisimilarity: two bisimilar states are guaranteed to be trace equivalent, but not vice versa.

- Trace equivalence: linear,
  bisimilarity: branching

Trace Equivalence vs. Bisimilarity

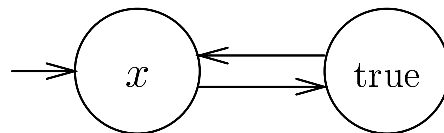


SAL versus STL

- Recall: bisimilarity is fully abstract semantics for STL
- Theorem: Trace equivalence is fully abstract semantics for SAL.
  - no SAL formula can distinguish between two states that are trace equivalent
  - for every two states that are not trace equivalent, there exists a SAL formula that is satisfied by only one of the two states
- Corollary: SAL is less distinguishing than STL
- Corollary: To solve model checking problems for SAL, we can first compute bisimilarity quotient

Expressive Power

- SAL and STL have incomparable expressive powers
- If s and t are trace-equivalent, but not bisimilar, then they agree on all SAL formulas, but STL can distinguish between them
- In general, SAL cannot express existential properties e.g. from every reachable state, some p state is reachable
- STL cannot count: no STL formula is equivalent to the automaton M_{even} (x holds in every even numbered state in every trajectory)



Hierarchical verification

- Top-down design approach:
  - Construct a high level design
  - debug using model checking
  - Add more details, and repeat
- Refinement checking: establishing conformance between successive levels
- What is the definition of refinement?
  - Bisimilarity: CCS
  - Failures containment: CSP
  - Trace containment: most common

Towards Trace Semantics

- Given a module P, which aspects of P determine completely its interaction with other modules?

- Communication interface of P:
  - the set of interface variables
  - the set of external variables
  - the awaits dependencies among the observable variables

- Interaction of P with other modules is completely determined by
  - Communication interface of P
  - Trace language L_P

- Traces of a compound module are completely determined by the traces of its components

- Let P and Q be compatible modules, and let $\overline{a}$ be a word over the observations of the compound module $P \,\|\, Q$. Then, $\overline{a}$ belongs to the language $L_{P \,\|\, Q}$ iff the projection $\mathsf{obs}X_P[\overline{a}]$ belongs to L_P and the projection $\mathsf{obs}X_Q[\overline{a}]$ belongs to L_Q.

Implementation Relation

- Our notion of refinement: P implements Q
- Intuitively: P is as complex as Q: P has possibly more interface and external variables than Q, P has more await dependencies among its observable variables, and has less traces than Q, and thus, more constraints on its execution.
- Def: P implements Q, denoted $P \preceq^L Q$, if
  - $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$,
  - $\mathsf{extl}X_Q \subseteq \mathsf{obs}X_P$,
  - if $y \prec_Q x$ then $y \prec_P x$, and
  - if $\overline{a}$ is a trace of P then the projection $\overline{a}[\mathsf{obs}X_Q]$ of $\overline{a}$ onto the observable variables of Q is a trace of Q.
- Trace equivalence: $P \simeq^L Q$, if $P \preceq^L Q$ and $Q \preceq^L P$.
- The implementation relation is a preorder

Implementation Examples

- Synchronous solutions are more constrained than asynchronous solutions:
  - SyncMutex $\preceq^L$ Pete, but not vice versa
- Deterministic strategies implement nondeterministic specifications:
  - Scheduler $\preceq^L$ NonDetScheduler, but not vice versa.
- Specification of a circuit can have many trace-equivalent implementations:
  - specification Sync3BitCounterSpec is trace-equivalent to implementation Sync3BitCounter

Nondeterministic Scheduler

module *Scheduler* **is**
  private $task_1, task_2 \colon \mathbb{N}$
  interface $proc \colon \{0, 1, 2\}$
  external $new_1, new_2 \colon \mathbb{N}$
  atom $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  init
    $\| \; true \rightarrow task_1' := new_1'$
  update
    $\| \; proc' = 1 \rightarrow task_1' := task_1 + new_1' - 1$
    $\| \; proc' \neq 1 \rightarrow task_1' := task_1 + new_1'$
  atom $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
    similar to $A3$
  atom $A6$ **controls** $proc$**reads** $task_1, task_2$
  init
    $\| \; true \rightarrow proc' := 0$
  update
    $\| \; task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0$
    $\| \; task_1 > 0 \rightarrow proc' := 1$
    $\| \; task_2 > 0 \rightarrow proc' := 2$

Deterministic Scheduler

module *Scheduler* **is**
  private $task_1, task_2 \colon \mathbb{N};\ p \colon \{1,2\}$
  interface $proc \colon \{0,1,2\}$
  external $new_1, new_2 \colon \mathbb{N}$
  atom $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  init
    $[\!] \ true \rightarrow task_1' := new_1'$
  update
    $[\!] \ proc' = 1 \rightarrow task_1' := task_1 + new_1' - 1$
    $[\!] \ proc' \neq 1 \rightarrow task_1' := task_1 + new_1'$
  atom $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
    similar to A3
  atom $A5$ **controls** $proc, p$ **reads** $task_1, task_2, p$
  init
    $[\!] \ true \rightarrow proc' := 0;\ p' := 1$
    $[\!] \ true \rightarrow proc' := 0;\ p' := 2$
  update
    $[\!] \ task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0$
    $[\!] \ p = 1 \wedge task_1 > 0 \rightarrow proc' := 1;\ p' := 2$
    $[\!] \ p = 1 \wedge task_1 = 0 \wedge task_2 > 0 \rightarrow proc' := 2$
    $[\!] \ p = 2 \wedge task_2 > 0 \rightarrow proc' := 2;\ p' := 1$
    $[\!] \ p = 2 \wedge task_2 = 0 \wedge task_1 > 0 \rightarrow proc' := 1$

3-bit Counter Specification

module $Sync3BitCounterSpec$ **is**
  interface out_0, out_1, out_2
  external $start, inc$
  atom controls out_0 **reads** out_0 **awaits** $start, inc$
    update
      $[] \; start' = 1 \rightarrow out_0' := 0$
      $[] \; start' = 0 \wedge inc' = 1 \rightarrow out_0' := \neg out_0$
  atom controls out_1 **reads** out_0, out_1 **awaits** $start, inc$
    update
      $[] \; start' = 1 \rightarrow out_1' := 0$
      $[] \; start' = 0 \wedge inc' = 1 \rightarrow out_1' := out_0 \oplus out_1$
  atom controls out_2 **reads** out_0, out_1, out_2 **awaits** $start, inc$
    update
      $[] \; start' = 1 \rightarrow out_2' := 0$
      $[] \; start' = 0 \wedge inc' = 1 \rightarrow out_2' := (out_0 \wedge out_1) \oplus out_2$

3-bit Counter Implementation

$$\begin{aligned}
&\mathit{Sync1BitCounter} \;=\; \textbf{hide}\ \mathit{set}, \mathit{reset}, z\ \textbf{in}\\
&\quad \text{—}\textbf{interface}\ \mathit{out}, \mathit{carry}\\
&\quad \text{—}\textbf{external}\ \mathit{start}, \mathit{inc}\\
&\quad \| \ \mathit{SyncLatch}[\mathit{set}, \mathit{reset}, \mathit{out}]\\
&\quad \| \ \mathit{SyncAnd}[\mathit{in}_1, \mathit{in}_2, \mathit{out} := \mathit{out}, \mathit{inc}, \mathit{carry}]\\
&\quad \| \ \mathit{SyncOr}[\mathit{in}_1, \mathit{in}_2, \mathit{out} := \mathit{carry}, \mathit{start}, \mathit{reset}]\\
&\quad \| \ \mathit{SyncNot}[\mathit{in}, \mathit{out} := \mathit{reset}, z]\\
&\quad \| \ \mathit{SyncAnd}[\mathit{in}_1, \mathit{in}_2, \mathit{out} := \mathit{inc}, z, \mathit{set}]
\end{aligned}$$

$$\begin{aligned}
&\mathit{Sync3BitCounter} \;=\; \textbf{hide}\ \mathit{carry}_0, \mathit{carry}_1, \mathit{carry}_2\ \textbf{in}\\
&\quad \text{—}\textbf{interface}\ \mathit{out}_0, \mathit{out}_1, \mathit{out}_2\\
&\quad \text{—}\textbf{external}\ \mathit{start}, \mathit{inc}\\
&\quad \| \ \mathit{Sync1BitCounter}[\mathit{inc}, \mathit{out}, \mathit{carry} := \mathit{inc}, \mathit{out}_0, \mathit{carry}_0]\\
&\quad \| \ \mathit{Sync1BitCounter}[\mathit{inc}, \mathit{out}, \mathit{carry} := \mathit{carry}_0, \mathit{out}_1, \mathit{carry}_1]\\
&\quad \| \ \mathit{Sync1BitCounter}[\mathit{inc}, \mathit{out}, \mathit{carry} := \mathit{carry}_1, \mathit{out}_2, \mathit{carry}_2]
\end{aligned}$$

Implementation Problem

- Input: two modules P and Q
- Output: Does $P \preceq^L Q$ hold?
- Checking conditions related to communication interface is easy
- Main obstacle: checking trace inclusion: if $\overline{a}$ is a trace of P then $\overline{a}[\mathsf{obs}X_Q]$ is a trace of Q
- Reduces to language-inclusion question:
  - Let $P' = \textbf{hide}\ (\mathsf{obs}X_P \setminus \mathsf{obs}X_Q)\ \textbf{in}\ P$
  - Check language-inclusion problem $(K_{P'}, K_Q)$

Complexity of Implementation Checking

- Doubly exponential for propositional modules
  - suppose P is propositional with k vars
  - suppose Q is propositional with ℓ vars
  - Language-inclusion test is $O(4^k \cdot 2^{2^\ell})$
- EXPSPACE-complete
- Special case: Q is observably-deterministic
  - Determinization of K_Q not needed
  - Language-inclusion test is $O(4^{k+\ell})$
  - PSPACE-complete

Checking Implementation

- Language inclusion test too expensive
- Two ways to simplify
- Simplify the implementation check by exploiting the structure of the module descriptions
  - Compositional reasoning
  - Assume guarantee reasoning
- Instead of implementation, check stronger requirements
  - Simulation relations
  - Refinement mappings

Compositionality

- If we prove that a module P implements another module Q, can we substitute P for Q in all contexts?

- The preorder $\preceq$ on reactive modules is compositional if for all modules P and Q, if $P \preceq Q$ then
  1. for every reactive module R that is compatible with P, R is compatible with Q and $P \| R \preceq Q \| R$;
  2. for variable x of P, $\mathbf{hide}\ x\ \mathbf{in}\ P \preceq \mathbf{hide}\ x\ \mathbf{in}\ Q$;
  3. for every variable renaming ρ, $P[\rho] \preceq Q[\rho]$.

- Theorem: The implementation preorder $\preceq^L$ on modules is compositional.

- A compositional equivalence on modules is called a module congruence.

- Trace equivalence is a module congruence

Compositionality Continued

- Compositionality is a basic soundness requirement of a concurrency theory
- For trace semantics (as in our case) it follows from
  - Implementation behaves logical implication (language inclusion)
  - Parallel composition behaves like logical conjunction (intersection of trace sets)

Using Compositionality

- Goal: To establish that a compound module $P_1 \| P_2$ implements the specification $Q_1 \| Q_2$

- We wish to avoid considering $P_1 \| P_2$, but consider them one at a time

- It suffices to establish that

  1. the component module P_1 implements Q_1,
  2. the component module P_2 implements Q_2

Simple Example

- Establish that synchronous message-passing protocol is an implementation of the asynchronous one.

- Module SyncMsg is

  $\textbf{hide}\ \mathit{ready}, \mathit{transmit}, \mathit{msg}_S\ \textbf{in}\ \mathit{SyncSender} \parallel \mathit{Receiver}$

- Module AsyncMsg is

  $\textbf{hide}\ \mathit{ready}, \mathit{transmit}, \mathit{msg}_S\ \textbf{in}\ \mathit{AsyncSender} \parallel \mathit{Receiver}.$

- To prove $\mathit{SyncSender} \preceq^L \mathit{AsyncSender}$, it suffices to establish $\mathit{SyncSender} \preceq^L \mathit{AsyncSender}$