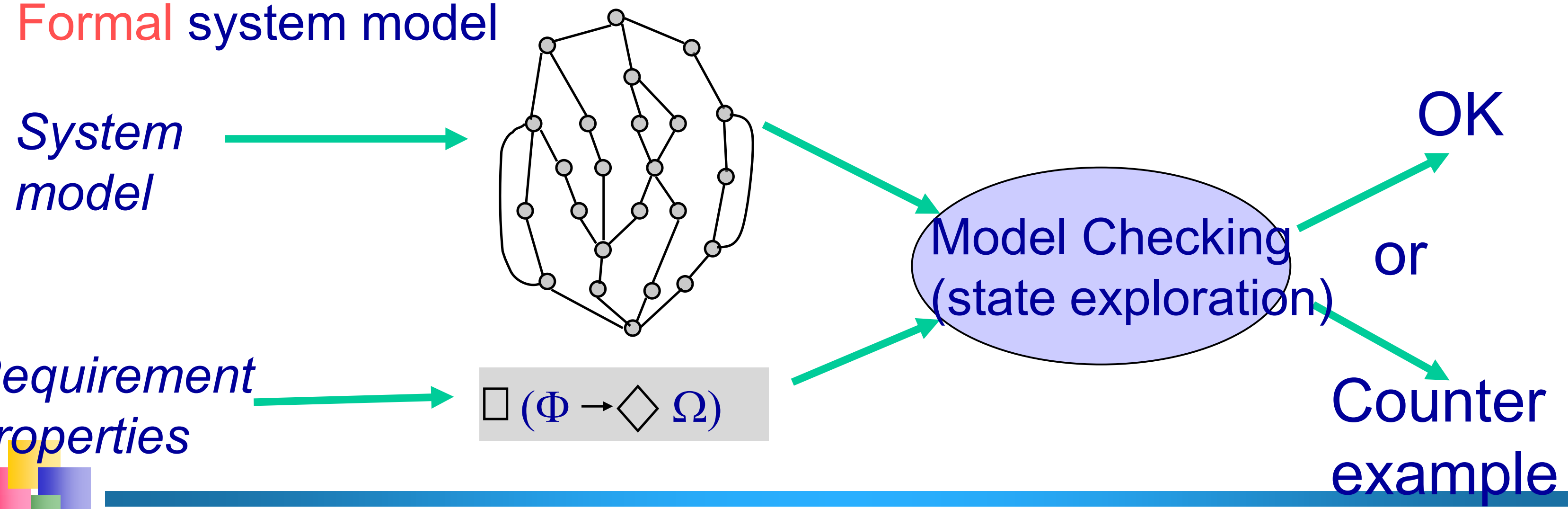
Introduction to Process Algebra

Moonzoo Kim
School of Computing
KAIST

Review of the Previous Class

- We have seen tragic accidents due to software and specification bugs
- These bugs are hard to find because those bugs occurs only in “exceptional” cases
- Informal system specification and requirement specification makes automatic analysis infeasible, which results in incomplete coverage
- To provide better coverage, we need
  - Formal requirement specification
  - Formal system model


System model
Requirement properties
$\square(\Phi \rightarrow \Diamond \Omega)$
Model Checking (state exploration)
OK
or
Counter example

Outline

- Requirement specification problems
- Viewpoint on “meaning”(semantics) of system
- Complexity of a system
- Formal modeling v.s. programming
- Introduction to process algebra

Requirement Specification Problems

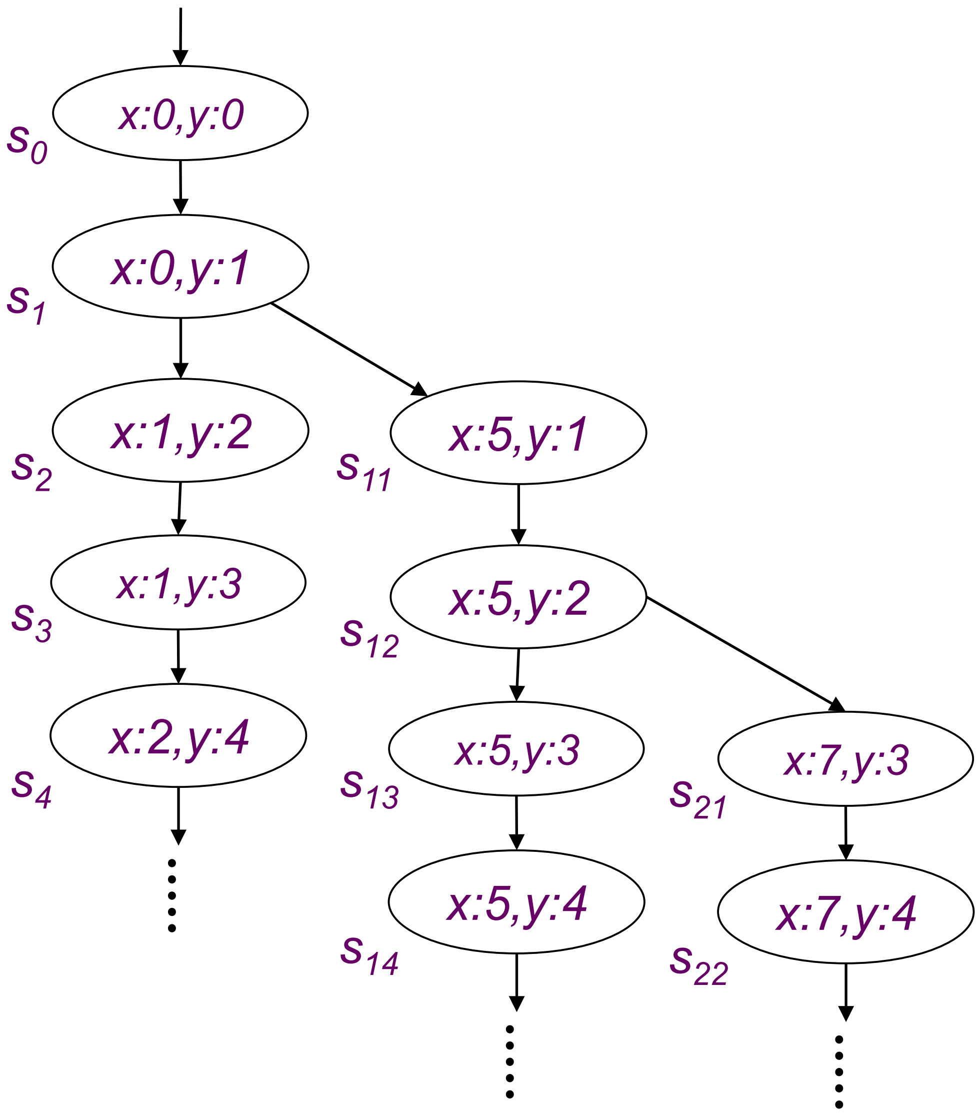
- Ambiguity
  - Expression does not have unique meaning, but can be interpreted as several different meaning.
    - Ex. `long` type in C programming language
- Incompleteness
  - Relevant issues are not addressed , e.g. what to do when user errors occur or software faults show.
    - Ex. See next slides
- Inconsistency
  - Contradictory requirements in different parts of the specification.

Example (retail chain management software)

- If the sales for the current month are below the target sales, then a report is to be printed,
  - unless the difference between target sales and actual sales is less than half of the difference between target sales and actual sales in the previous month
  - or if the difference between target sales and actual sales for the current month is under 5 percent.

Viewpoint on Semantics of a System

- A system execution σ is a sequence of states $s_0 s_1 \ldots$
  - A state has an environment ρ_s:*Var-> Val*
- A system has its semantics as a set of system executions

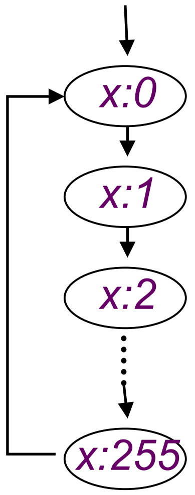


Complexity of Systems

- The complexity of a system is sometimes more accurately expressed using semantic viewpoint (# of reachable states) rather than syntactic viewpoint (line # of source code)
  - the number of different *states* a system can reach
    - Ex> An integer has 2^{32} (~4000000000) possible values

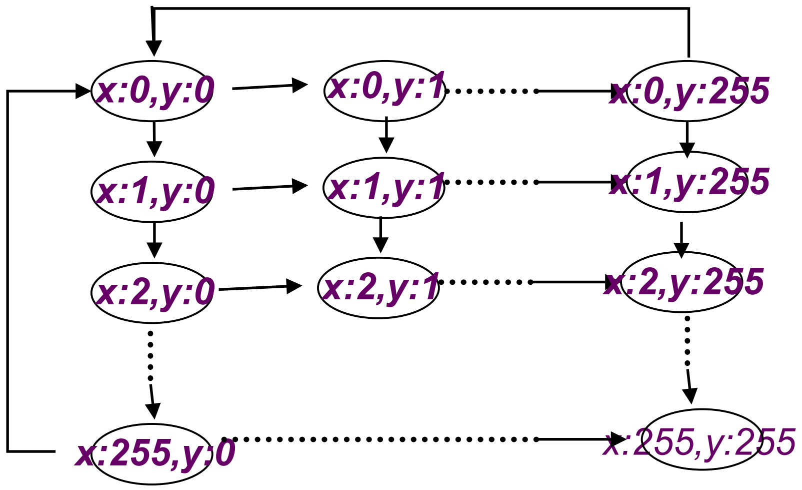
Example

```
active type A() {
byte x;
again:
    x++;
   goto again;
}
```



```
active type A() {
byte x;
again:
    x++;
   goto again;
}

active type B() {
byte y;
again:
    y++;
   goto again;
}
```



Formal Modeling V.S. Programming

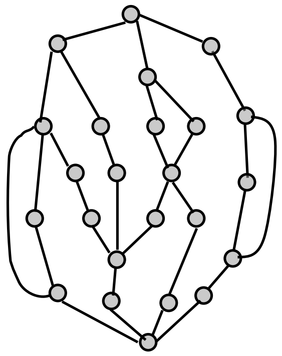
		Formal Modeling	Programming
Static Aspects	Abstraction Level	High	Low
	Development Time	Short	Long
Dynamic Aspects	Executable	Yes (model checking) No (theorem proving)	Always
	System Semantics	Mathematically defined	Usually given by examples
	Environment Semantics (i.e. testbeds)	Mathematically defined	Usually given by examples
	Program State Space	Manageable (i.e. tractable state space)	Unmanageable (i.e. beyond computing power)
	Validation	By exhaustive exploration or deductive proof	By testing (incomplete coverage)

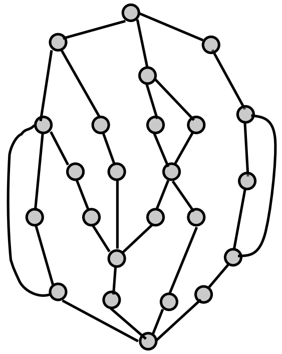
Complex System Attributes

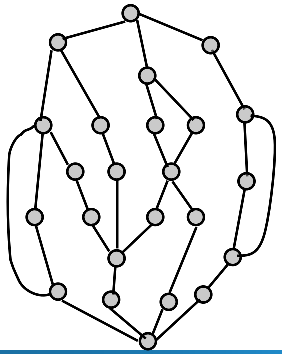
- You may not need to model a simple system such as +,*, or HelloWorld.
- However, you must have a scientific way of abstracting/modeling a system with complex structure, e.g.,
  - Hierarchy
  - Concurrency
  - Communication
- Also, you need to have a systematic way to analyze the correctness of your design

How to Define "Correctness" ?

- System design vs. Requirement spec.

1)  $\models$ $\Box(\Phi \rightarrow \Diamond\, \Omega)$

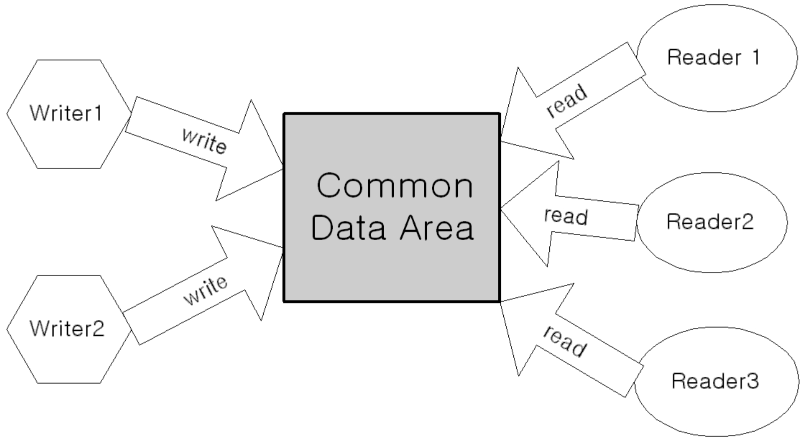
2)  $\Dashv$ $\Box(\Phi \rightarrow \Diamond\, \Omega)$

3)  $\approx$ $\Box(\Phi \rightarrow \Diamond\, \Omega)$

Requirement Specification

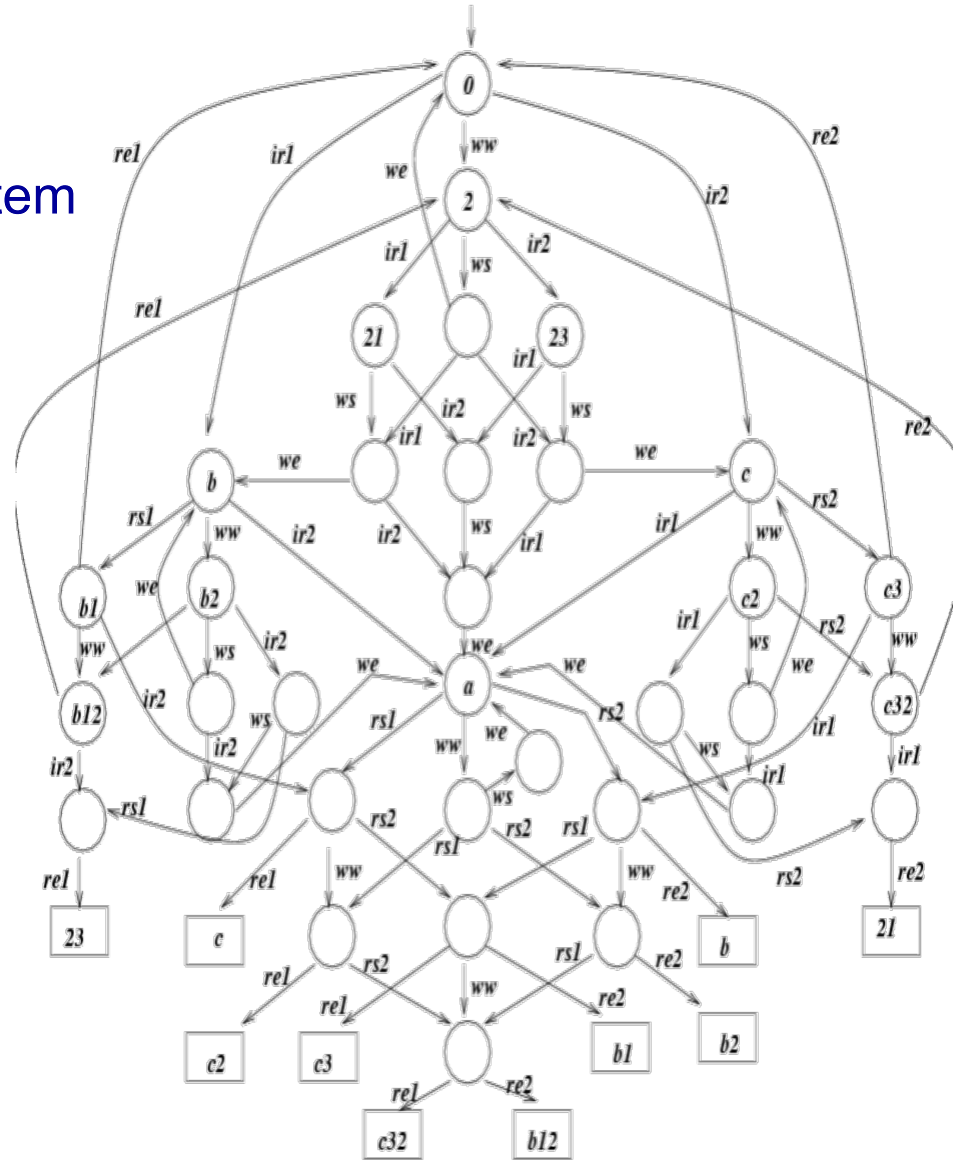
- Requirement specifications are the goals that a target software must satisfy
  - Ex. For a system containing 3 readers, 2 writers, and common common data area, the system should satisfy the following three requirement properties

- *Concurrency (CON)*
  - *Multiple readers can read data concurrently*
- *Exclusive writing (EW)*
  - *A writer can write into the data area at an instant with no readers*
- *High priority of a writer (HPW)*
  - *A writer's request should have a higher priority than that of a reader*



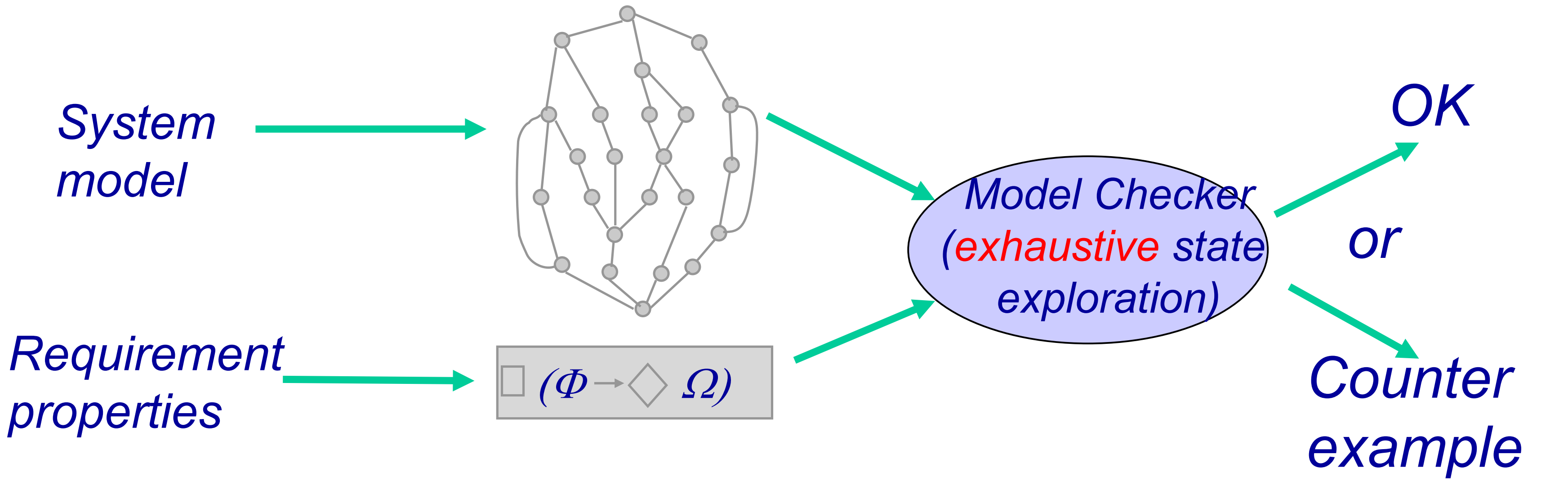
System Design Model

- Abstract description of a target system
- Model must have clear meaning/semantics
- Ex. A system design model for 2 readers and one writer in process algebra
  - RW system has 9 events
    - {ir1,rs1,re1,ir2,rs2,re2,ww,ws,we}
    - incoming reader 1, reader 1 starts, reader 1 ends
    - waiting writer, writer starts, writer ends
  - System = $(R_1| R_2| W|D)\backslash\{\ldots\}$
    - R_i=check_lock. read_request$_i$. start_read$_i$. end_read$_i$.R_i
    - W_i= check_lock. write_request. start_write. end_write.W
    - D = 'read_request$_1$. …



Design Analysis

- We definitely want to guarantee/prove that a system design model *M* satisfies a requirement property ϕ
  - $M \models \phi$


System model
Requirement properties
$\square(\Phi \rightarrow \diamond \Omega)$
Model Checker (exhaustive state exploration)
OK
or
Counter example

Necessity of Rigorous Req. Spec.

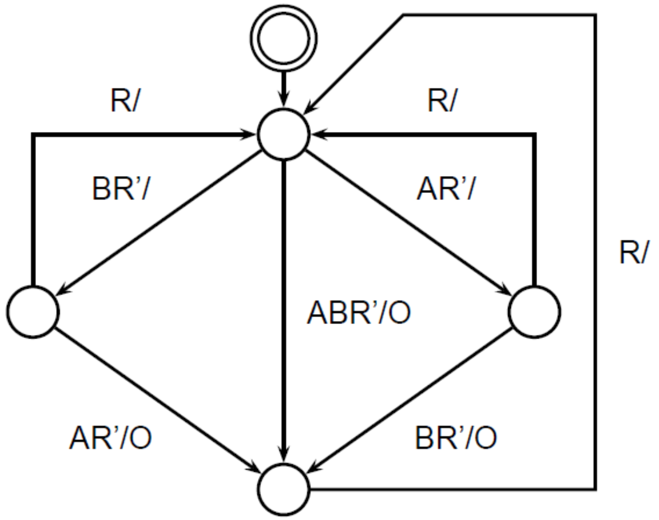
- Specification in natural languages can be easily incomplete due to assumption of implicit "common senses" which actually do not exist
- Ex. For the 3-readers and 2-writers system
  - Concurrency (CON): "Multiple readers can read data concurrently."
    - What if only 2 readers can read concurrently?
  - Exclusive writing (EW): "A writer can write into the data area at an instant with no readers"
    - What if two writers write into the data area at the same time?
  - High priority of a writer (HPW): "A writer's request should have a higher priority than that of a reader"
    - What if a writer requests one second later than a reader? Should the system wait for handling readers request? If so, how long?

Necessity of Rigorous Req. Spec.

- Suppose that R_i means i th reader is reading.
- The requirement ϕ_{CON} for concurrency can be written in propositional logic
  - If it is ok for two readers to read concurrently, ϕ_{CON} should be
    - $(R_1 \wedge R_2) \vee (R_2 \wedge R_3) \vee (R_3 \wedge R_1)$ for some time instant t
    - Note that if it is ok for only two readers to read concurrently, $\phi_{CON} = (R_1 \wedge R_2 \wedge \neg R_3) \vee (\neg R_1 \wedge R_2 \wedge R_3) \vee (R_1 \wedge \neg R_2 \wedge R_3)$
  - If all three readers should be able to read concurrently, ϕ_{CON} should be
    - $(R_1 \wedge R_2 \wedge R_3)$ for some time instant t

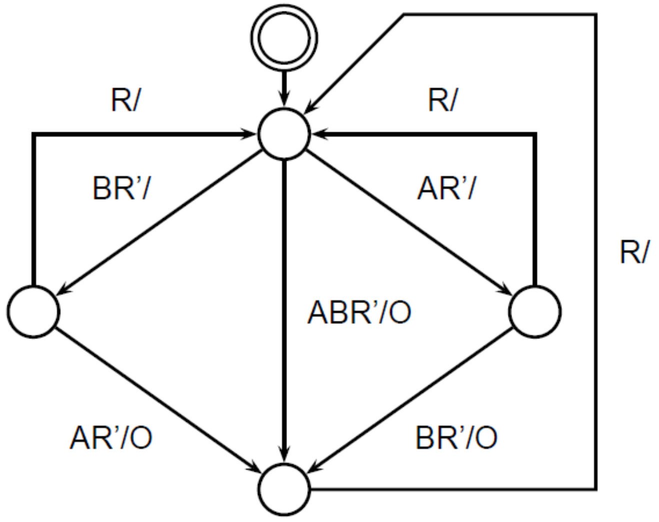
Another Example Model (1/2)

- The req. spec.
  - The output O should occur when inputs A and B have both arrived.
  - The R input should restart this behavior.
- System design.



Implementation of the Example (2/2)

- System design.



- Implementation in a synchronous language Esterel

```
module ABRO_example:
input A, B, R;
output O;
loop
    [ await A || await B ];
    emit O
each R
end module
```

The implementation is simple since language includes notions of signals, waiting, and reset, etc.

Process Algebra

- A process algebra consists of
  - a set of operators and syntactic rules for constructing processes
  - a semantic mapping which assigns meaning or interpretation to every process
  - a notion of equivalence or partial order between processes
- Advantages: A large system can be broken into simpler subsystems and then proved correct in a modular fashion.
  - A hiding or restriction operator allows one to abstract away unnecessary details.
  - Equality for the process algebra is also a congruence relation; and thus, allows the substitution of one component with another equal component in large systems.
- Note that the model is constructed in a component-based way, but the analysis is not.

Calculus of Communicating Systems (CCS)

- Developed by R.Milner (Univ. of Cambridge)
  - ACM Turing Awardee in 1991
- Provides many interesting paradigms
  - Emphasis on communication and concurrency
    - Provides compact representation on both communication and concurrency
      - Ex> a (receive) and a' (send)
      - Ex> | (parallel operator)
  - Provides observation based abstraction
    - Hiding internal behaviors using \ (restriction) operator, i.e., considering all internal behaviors as an invisible special action τ
  - Provides correctness claim based on equivalence
    - Branching time based equivalence
      - Strong equivalence v.s. weak equivalence

Overview on CCS Syntax and Semantics

- CCS describes a system as a set of communicating Processes
- Behavior of a process is expressed using actions
  - Act =input_actions U output_actions U {τ}
- Each process is built based on the following 7 operators
  - Nil (null-ary opeartor): 0
  - Prefix: a.P
  - Definition: P = a.b.Q
  - Choice: a.P + b.P
  - Parallel: P | Q
  - Restriction: P \ {a,b}
  - Relabelling: P[a/b]
- Each operator has a clear formal semantics via inference rules (premises-conclusion rules)
  - Based on these inference rules, a meaning/semantincs of a process is given as a labelled transition system

Example of a CCS System

- A set of actions Act = {a,a’,b,τ}
- We define a CCS system Sys as
  - Sys = (a.E + b.0) | a’.F
- Sys can executes one of the following 4 actions
  - Sys –a-> E | a’F
  - Sys –a’-> (a.E + b.0)|F
  - Sys –b-> 0 | a’.F
  - Sys - τ-> E|F

$$
\dfrac{\dfrac{\dfrac{}{\text{a.E –a-> E}}\ \text{Prefix}}{\text{(a.E + b.0)) –a-> E}}\ \text{Choice}_L}{\textbf{(a.E + b.0)) | a’.F –a-> E | a’.F}}\ \text{Par}_L
$$

$$\begin{tikzcd}
 & & \text{Sys =(a.E + b.0) | a’.F} \arrow[dll, "a"'] \arrow[dl, "a'"'] \arrow[d, "b"] \arrow[dr, "\tau"] & \\
\text{E | a’.F} \arrow[rrr, bend right=60, "a'"'] & \text{(a.E + b.0)|F} \arrow[d, "b"'] \arrow[rr, bend right, "a"] & \text{0 | a’.F} \arrow[dl, "a'"] & \text{E | F} \\
 & \text{0 | F} & &
\end{tikzcd}$$