EECS 249B

Joon Kim

System Design

CPS: Co-design of physical system and controller, discrete & cont.

↳ How to control complexity to the right level of abstraction?

ex) "New DNA" Vehicles: electrical, connected, ML, autonomous, ...
V2V / V2X communications, may be critical to Lv5 autonomy?

ex) Bio-Cyber Systems: Our bodies as machines? Brain signals?
Synthetic Biology, can we program biology into higher-order systems?
Physically Constrained AI, Alpha fold (2)!

What are the design challenges? → confirmation biases! certification robustness! automating certification!

→ Safety: Tesla recall on self-driving software, Boeing crashes, Toyota "sticky pedal" recall & investigation on software.
↳ There are too many potential cases to test → Actually infeasible

→ Security: Attack surface is greatly expanding! Adversarial
↳ Interception, Modification, Fabrication, Interruption, ...
↳ Desired: Confidentiality, Integrity, Availability, Authentication, ...

Embedded vs "Ordinary" Software: Limited resources (time, space), compiler must respect limits.... but programming abstracts this level!

↳ The standard trick is to fall back to plan B resources on failure.

But embedded software needs to know the exact, worst case verification on a (potentially) different set of abstractions.

↳ Turing Machines terminate, but embedded softwares shouldn't!

Concurrency Problem: Humans are not good at conceiving concurrency, how do we verify the correctness of such systems?

↳ Deadlocks, Priority Inversion, Nondeterminism, ... Incomprehensible!

Hardware have also become parallel over the years.

Equation-Based Model: Acausal
Finite State Machine: Untimed
⇒ Softwares that are imperative

Traditionally, complexity has been handled by decomposition & abstraction.

"Construction" also manages complexity: constraining, refinement.
↳ regularizing some rules

Not "Freedom of Choice", but "Freedom from Choice"!

Model-Based Design

Model-Based Design: Create a mathematical model of all the parts of the cyber-physical system. Construct the implementation from the model. Ideally, we can automate the construction (like a compiler)

[Gdomb, '68] "You never strike oil by drilling through the map"
↳ But still, this does not diminish the value of the map (model)

Advantages: Time & Cost effective, Optimization possible
Difficulties: real plants are not reliably deterministic, experimental validation is still widely used, insufficient investments & methodologies

Transistor-Level Design Flow: Manual Design → SPICE Verification
Gate-Level " : Logic Simulator! Abstracts waveforms → 0/1
↳ 100x speedup vs SPICE!
Register Transfer Level (RTL): Abstract away circuit delays, comb. logic
↳ Hardware Design Language (HDL) → RTL synth. → logic optim. → physical

⇒ Increase in efficiency / engineer due to high-level abstraction!

Synthesis ⇄ Verification are intimately connected

Verification: "Is what I think (HDL) I want what I really (spec) want?"

- Formal Verification: symbolically checks model/theorems
- Simulation: apply simulation stimulus to model
- Emulation: implement a version of circuit on emulator
- Rapid Prototyping: create prototype of actual hardware

↑ exhaustive
↓ high-fidelity

Model Checking: Given a property and design, output Yes or counter ex.

Software Simulation: driver(input vectors) → model(HDL) → monitor(Y/N)

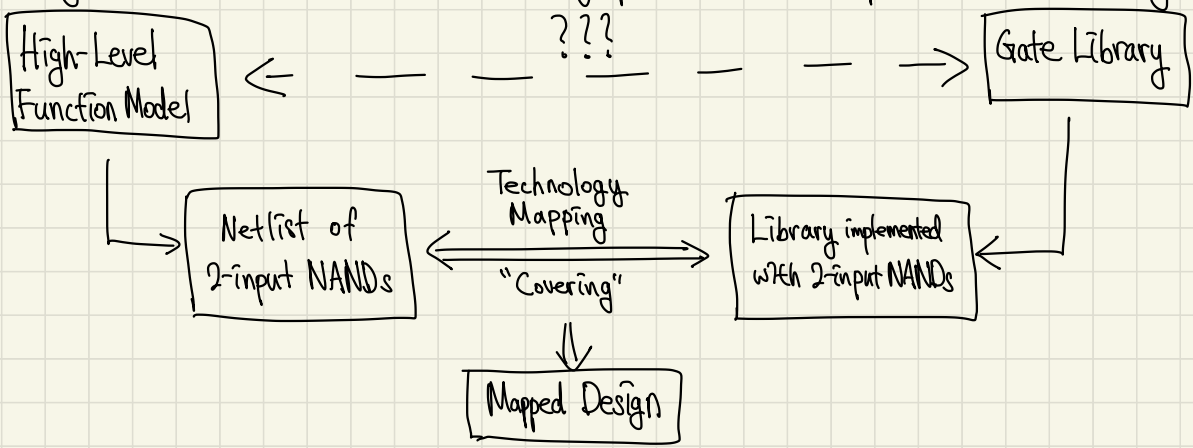
Then, Implementation Verification & Manufacture Verification

Verification is now the new bottleneck; upto 70% of design efforts

- When is the verification "enough"?
- Verification → Robustness? Did we capture all kinds of bugs?
- Verification → Repair? Can we get information on how to fix the bug?
- Computational Engine Efficiency/Scalability; SMT solvers?
- Verifying analog/mixed signal systems?

Synthesis: How do we achieve what we want while being optimal?

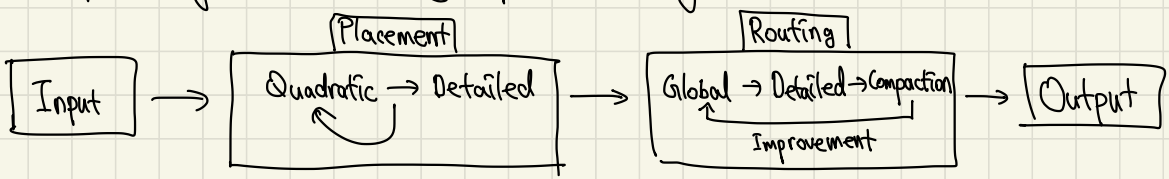
Logic: How do we resolve the gap between the spec and the library?



- Separation of function and architecture
- Common language for functional & architectural level netlists
  ↳ Boolean logic, NAND2 gates
- Automated Mapping!

Physical Design: How do we place the physical gates optimally?

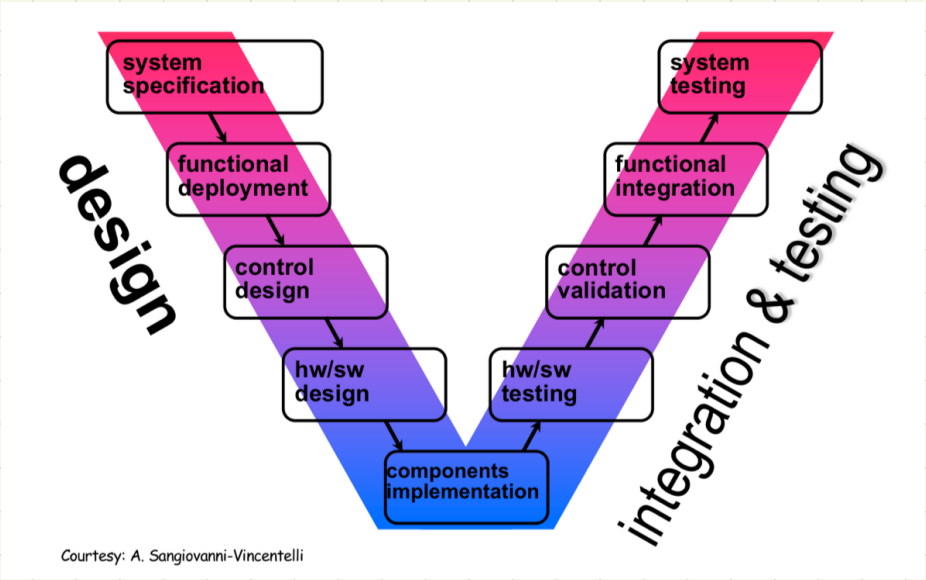
↳ "Optimality" means wiring, space, energy, heat, etc...



IC Design is getting more expensive as chips are getting smaller!

→ We started to think of reusing bigger chunks (IP-Based Design)

The V Design Process: A "Linear" approach → ... what can go wrong?

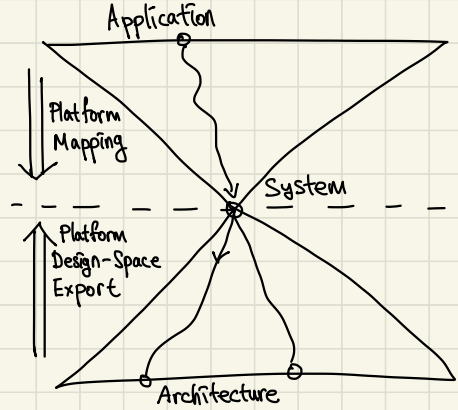


↳ This still alludes to a "waterfall" issue of potential expensive redesign.

PBD/CBD

Separation of Concerns: Keep the What separated from the How

↳ Platform-Based Design → Meet-in-the-Middle



Top-Down: Spec → solution components

Bottom-Up: Components → abstractions

Limits the exploration space for reasonable results in limited time, may be incremented and/or reused for different applications!

Def) Platform: library of components that can be assembled to generate a design at that level of abstraction.

↳ characterized in terms of performance parameters ("quantities") & supported functionalities.

The platform is a parameterization of solution space!

Large layers can produce large gaps from the lower bound of prediction.

Design Flow (Theory): Initial intent captured with declarative notations (≃ interface in Java)

↳ at this stage, HW/SW is not decided yet.

Classical (Decoupled) design: Control and Implementation spaced are separate.

Ideally, all parameters should be considered all at once, but too complex.

Implementation Abstraction Layers exposes non-idealities that could affect the performance of the controlled plant. (delay, error, imprecision)

↳ as long as these are obeyed, we can design higher layers!

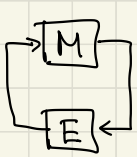
Contracts: How to check/enforce consistency between two layers??

↳ Components, Composability, Composition, Environment

(machine M) (typings) (product X) (E s.t. E×M is closed)

↳ open or closed

↳ commutative & associative!



Assume/Guarantee Contract: 3-tuple of sets $(\vec{S}, A, G)$ ($\vec{S}$ → variables)

ex) $C_1 := ((\text{input: } x,y, \text{ output: } z) \in \mathbb{R},\ y \neq 0,\ z = x/y)$.

An implementation M satisfies, i.e. $M \models (A,G)$ iff M refines G in the context of A

↳ $M \cap A \subseteq G$. (equivalently, $M \subseteq G \cup \neg A$ by propositional logic)

Environment E is valid, $E \models_E (A,G)$ iff $E \subseteq A$.

* $M \models (A,G)$ iff $M \models (A,G')$ where $G' := G \cup \neg A$ (saturated form)!

(A,G') is consistent iff $G' \neq \emptyset$, compatible iff $A \neq \emptyset$.

Composition: If I take any implementation of C_1 and C_2, what is the minimum contract that I can say about them? $(C_1 \otimes C_2)$

- must guarantee what the components guarantee $(G_1 \cap G_2)$
- must accept what is accepted by both components $(A_1 \cap A_2)$
- however, part of the assumptions of one can be discharged by other!
- look at the weakest assumption assuring both initial assumptions are met

$$G_{C_1 \otimes C_2} = G_{C_1} \wedge G_{C_2} \qquad A_{C_1 \otimes C_2} = \max_A \{A \mid \begin{matrix} A \wedge G_{C_2} \Rightarrow A_{C_1} \\ A \wedge G_{C_1} \Rightarrow A_{C_2} \end{matrix}\}$$

$$\Longleftrightarrow$$

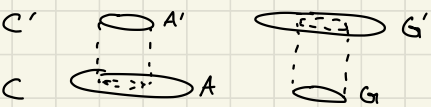
$$G_\otimes = G_1 \wedge G_2 \qquad A_\otimes = (A_1 \wedge A_2) \vee \neg G_1 \vee \neg G_2$$

Two contracts are compatible when guarantees of one does not violate the assumptions of the other / environment enforces condition

Def) Alphabet Equalization: Given P assertion over $\Sigma, \Sigma' \subseteq \Sigma, \Sigma'' \supseteq \Sigma$, projection $pr_{\Sigma'}(P)$ is the set of all restrictions to Σ' of all behaviors belonging to P, inverse projection $pr^{-1}_{\Sigma''}(P)$ is the set of all behaviors over Σ'' that project to Σ as behaviors of P. Then, AE of (Σ_1, P_1) and (Σ_2, P_2) are $(\Sigma, pr^{-1}_{\Sigma''}(P_i))$ for $i=1,2$ where $\Sigma = \Sigma_1 \cup \Sigma_2$.

$\preceq$

Refinement: $C=(A,G)$ is "stronger" than $C'=(A',G')$ if it satisfies:



- $G \subseteq G'$ and $A' \supseteq A$
- If M satisfies C, then M also satisfies C'
- If E satisfies C', then E also satisfies C

Conjunction: the greatest lower bound of a refinement pre-order

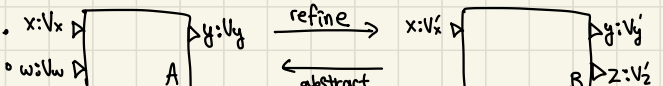
- $C_1 \wedge C_2 = (A_1 \cup A_2, G_1 \cap G_2)$ ↳ combining different viewpoints of the same system
- If M satisfies both C_1 and C_2, then M satisfies $C_1 \wedge C_2$

* Refinement is compositional! $C_1 \preceq C_1', C_2 \preceq C_2' \rightarrow C_1 \otimes C_2 \preceq C_1' \otimes C_2'$

ex) Function Types: double foo(int bar) $\geq$ int foo(double bar)

(More restrictive G; less restrictive A)

ex) Type Refinement for Actors:



$P_B \subseteq P_A, Q_A \subseteq Q_B, \forall p \in P_B, V_p \subseteq V_p'$ $P_A = \{x, w\}$ $Q_A = \{y\}$ $P_B = \{x\}$ $Q_B = \{y, z\}$

$\forall q \in Q_A, V_q' \subseteq V_q$.

Independent Refinement: $\forall$ contracts C_1, C_2, C_1', C_2', if C_1 & C_2 are compatible and $C_1' \preceq C_1$ & $C_2' \preceq C_2$, then C_1' & C_2' are compatible and $C_1' \otimes C_2' \preceq C_1 \otimes C_2$.

↳ sometimes, we might want to reason on more abstract contracts because they are easier to manipulate, without hurting generality of the original!

$C_1 \wedge C_2 \preceq C_1$ and $C_1 \wedge C_2 \preceq C_2$.

$\forall C$, if $C \preceq C_1$ and $C \preceq C_2$, then $C \preceq C_1 \wedge C_2$.

Models of Computation

Def) Models of Computation: a syntax and semantics to specify rules for computation, communication, concurrency, etc.

ex) Finite state machines, Turing machines, Differential equations, ...

↳ should unambiguously capture functionality, synthesis, verification.

↳ should consider expressibility, generality, simplicity, compilability, verifiability.

⇒ combining various MoCs can achieve desirable features!

Control VS Data Flow: time of arrival vs regularly streamed values

FSMs: Functional decompositions into states of operation.

↳ typically used in control functions / protocols (telecom, computers, ...)

can have synchronous or asynchronous communication

* Nondeterministic FSM are equivalent to FSMs! [Rabin '59]

Advantages: easy to use, powerful algorithms for synth/verif.

Disadvantages: sometimes overspecified, # states intractable, not numerical (not compact)

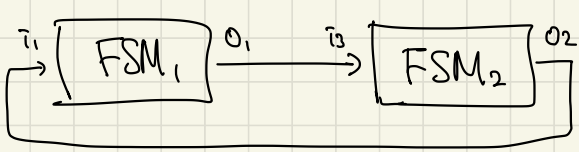
Def) FSM: tuple $(I, O, S, r, \lambda, \delta)$ where $\lambda \in (I \times S \times S)$, $\delta \in (I \times S \times O)$.

(I: input, S: state, λ: transition fn, O: output, r: initial state, δ: output fn.)

Composition of FSM: synchronocity? Assume instantaneous signals ("synchronous hypothesis")

↳ Given M_1 and M_2, find M' given constraint $C = \{(o, i_1, \ldots, i_n) \mid o \to i_j\}$.

What if there is a cycle? ex) $\lambda_1(\{i_1\}, S_1) = \{o_1\}$, $o_2 \notin \lambda_2(\{i_3\}, S_2)$?



Moore machines are well-defined.

Mealy machines can be <u>undefined</u>!

* Moore and Mealy are computationally (almost) identically powerful.

Moore: non-reactive (delayed by 1 cycle), easy composing & implementing

Mealy: reactive (no delay), hard to compose (causality loops) & implement

↳ how do we ensure the "synchronous hypothesis"?

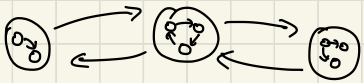
Synchronous is good for analysis, but real systems might not follow it! (multi-rate/distributed/heterogeneous)

Asynchronous FSM: Free to proceed independently, does not execute transitions at the same time, but does share a notion of time

→ Blocking read/write can give some level of determinism (don't change when reading/writing)

→ Rendez-vous synchronizes in some specific points to exchange data

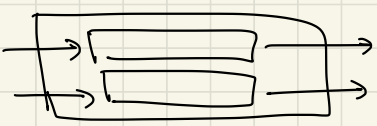
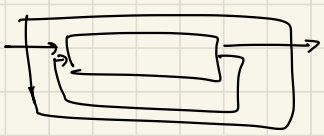
--- and many more possible choices!



Hierarchial FSM: each state "encloses" a smaller FSM of "or-states" and synchronous concurrency implies "and-states" (Statecharts)

↳ still has circular dependencies, not truly synchronous

Synchronous Reactive Model: all reactions are synced and instantaneous

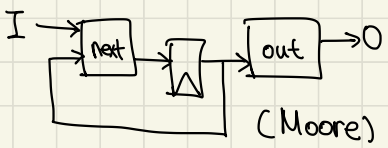
Parallel:  Feedback: 

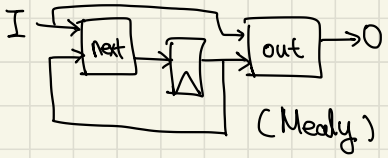
* Everything can be concieved with parallel-then-feedback composition!

How do we resolve feedback dependencies?

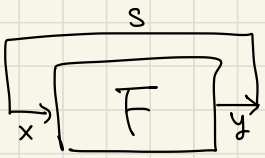
Think of the fixed point problem, $x = f(x)$. ← sometimes, this is solvable.

↳ Moore machines are easy to resolve. Mealy machines are more subtle.





We seek $s \in S^{\mathbb{N}}$ s.t. $F(s) = s$. Ideally, s exists and is unique.



$V_y \subseteq V_x$, $s : \mathbb{N} \to V_y \cup \{\text{absent}\}$. F is the actor function which for deterministic systems has form

$F : (\mathbb{N} \to V_y \cup \{\text{absent}\}) \to (\mathbb{N} \to V_y \cup \{\text{absent}\})$.

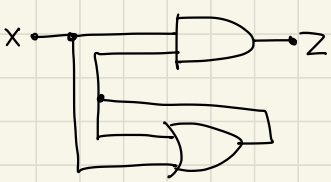
At the n-th reaction, if $\exists f(n) : (V_x \cup \{\text{absent}\}) \to (V_y \cup \{\text{absent}\})$ s.t. $s(n) = (f(n))(s(n))$, this is also a fixed point.

If the fixed point does not exist or is not unique, system is ill-formed.

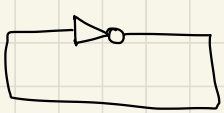
Otherwise, we say that the system is well-formed.

→ How do we determine whether a system is well-formed?

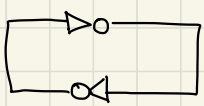
Intuition: cyclic circuits. Can we have well-behaving looped circuits?



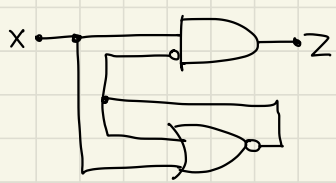
(well-formed)



(no solution)



(two solutions)



→ input x & output z are deterministic, but y?

Constructiveness guarantees no oscillations!

* Constructive ⇒ Well-formed, but not the converse.

We can test for constructiveness by "uncovering" unknown values.

start with $\vec{S}(n)$ unknown. determine about $(f(n))(\vec{S}(n))$ as best as possible. plug in any uncovered values in $\vec{S}(n)$ and iterate until no more values are uncoved (non-constructive) or we uncover all signals (constructive).

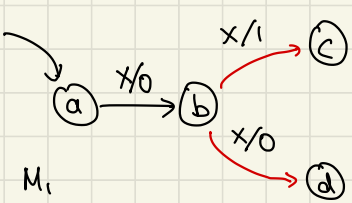
* but there is no $O(poly(n))$ algorithm where $n = |input\ domain|$.

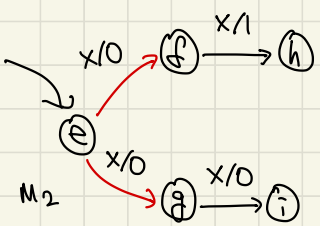
Temporal Logic

Can we say that one FSM "refines" another? → "Matching Game"

↳ $M_1 \geq M_2$ means that M_1 simulates M_2

Define a set S of tuples (s_2, s_1) where $s_1 \in M_1, s_2 \in M_2$.





$S = \{(e,a), (f,b), (g,b), (h,c), (i,d)\}$ → M_1 can follow behavior of M_2 $\forall s_2$!

* $M_1 \geq M_2 \wedge M_1 \leq M_2$ does not imply an equivalence relation!!!

↳ we need the condition that $S_{M_1,M_2} \longleftrightarrow S_{M_2,M_1}$ has the bijection $(a,b) = (b,a)$.

only then can we truly say that M_1 and M_2 are just syntactically different but is the same machine (equivalence)

How do we turn informal requirements to formal specifications?

↳ For timing-related systems, LTL & STL can capture their behaviors

ex) $GFp \Rightarrow$ infinitely often p, $G(p \to Fq) \Rightarrow$ "protocol of $p \rightsquigarrow q$"

LTL (formally): how do we define the satisfying object $\sigma \models \phi$?

↳ an infinite trace of Kripke structure (P, S, S_0, L, R), $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$

where $\sigma_i \subseteq P \ \forall i$. ex) $\sigma = \{p\}, \{q\}, \{\}, \{p, q\}, \ldots$

Let $\sigma[i..] := \sigma_i \sigma_{i+1} \sigma_{i+2} \ldots$. Satisfaction is defined recursively as:

$\sigma \models p \iff p \in \sigma_0$. → only for current step!

$\sigma \models \phi_1 \wedge \phi_2 \iff \sigma \models \phi_1 \wedge \sigma \models \phi_2$.

$\sigma \models \neg\phi \iff \sigma \not\models \phi$.

$\sigma \models G\phi \iff \forall i = 0, 1, \ldots, \sigma[i..] \models \phi$.

$\sigma \models F\phi \iff \exists i = 0, 1, \ldots, \sigma[i..] \models \phi$.

$\sigma \models X\phi \iff \sigma[1..] \models \phi$.

} → suffix-based!

$\sigma \models \phi_1 U \phi_2 \iff \exists i = 0, 1, \ldots, \phi[i..] \models \phi_2 \wedge \forall 0 \le j < i, \sigma[j..] \models \phi_1$.

→ both quantifiers!

STL: $\varphi := \mu \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi U_{[a,b]} \psi$ where $\mu: \mu(x) > 0$.

$(x, t) \models \mu \iff \mu(x[t]) > 0$.

$(X,t) \models \varphi \wedge \psi \iff (X,t) \models \varphi \wedge (X,t) \models \psi$

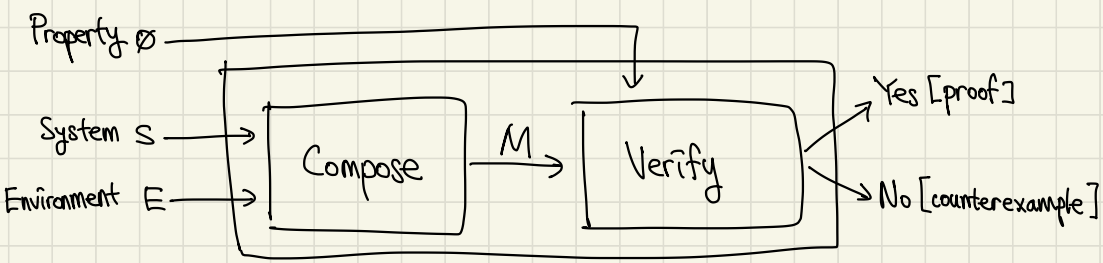
$(X,t) \models \neg\varphi \iff \neg((X,t) \models \varphi)$

$(X,t) \models \varphi U_{[a,b]} \psi \iff \exists t' \in [t+a, t+b] \text{ s.t. } (X,t') \models \psi \wedge$
$\forall t'' \in [t,t'), (X,t'') \models \varphi.$

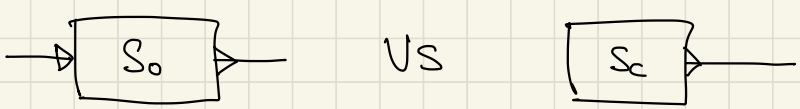
Model Checking

Reachability Analysis: find set of reachable states for a system

Model Checking: algorithm that determines if a system satisfies a temporal logic



Open vs Closed System:



FV requires systems to be closed, so we assume some environment.

Transition Systems: States + Transitions (+ Labels)

* possibly infinite sets of states/transitions!

Labeled Transition Systems: (Σ, S, S_0, R), $R \subseteq S \times \Sigma \times S$

Kripke Structure: (P, S, S_0, L, R), $L: S \to 2^P$, $R \subseteq S \times S$

↳ a path is a (potentially infinite) $s_0, s_1, s_2, \ldots$ s.t. $s_0 \in S_0$, $\forall i, (s_i, s_{i+1}) \in R$.

↳ a trace is the corresponding sequences of sets of labels $\sigma = L(s_0), L(s_1), \ldots$

Reachable State: $s \in S$ s.t. $\exists$ finite path $s_0, s_1, \ldots, s_n$ s.t. $s_n = s$.

Reachable State Space: subgraph that only contains reachable states

Deadlock State: no outgoing transition, $\nexists s'$ s.t. $(s, s') \in R$.

Linear-Time vs Branching-Time:

Consider this property: "it is possible to recover from any fault."

LTL can only encode "it will always recover from any fault."

* For finite state systems, state-space exploration can be fully automated!

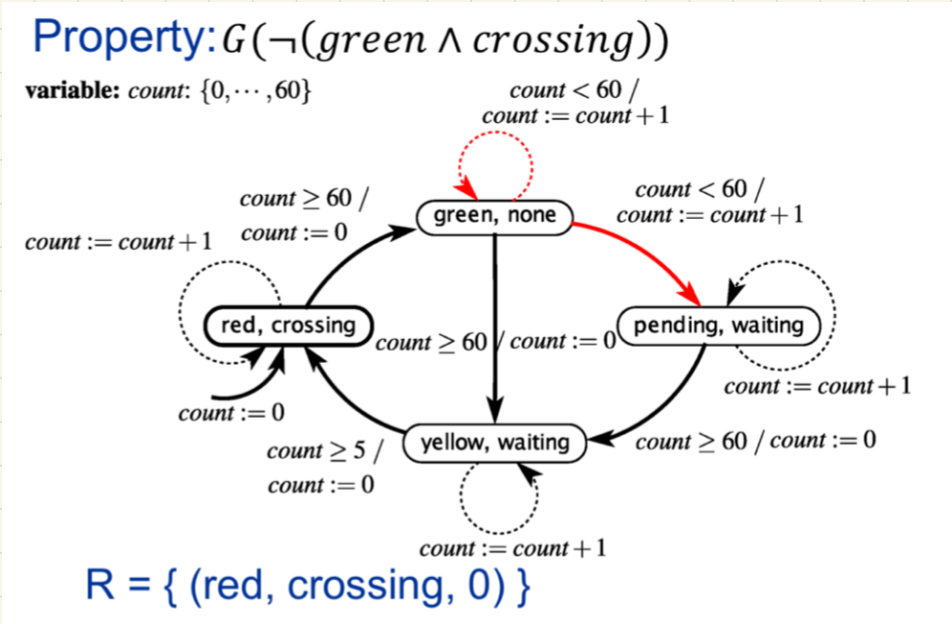
(although only in principle due to physical constraints)

Model Checking $G(p)$: enumerate all reachable states & check p es vs.

↳ we can use graph traversal!

Explicit Model Checking Example: property $G(\neg(\text{green} \wedge \text{crossing}))$

↳ we enumerate all reachable states to find that property holds!



Symbolic Methods: explore new sets of reachable states

↳ this requires some novelty in data representation, as boolean functions!

→ a finite set of states $S \mapsto f_S(x)$ s.t. $f_S(x) := \mathbb{1}\{x \in S\}$.

ex) $\Omega_1 = \{p,q,r \in \{0,1\}\}$. $S_1: p \vee q = \{pqr, pq\bar{r}, \bar{p}qr, \cdots\}$. 6 states.

$\Omega_2 = \{x \in \mathbb{R}, i \in \mathbb{Z}^+, b \in \{0,1\}\}$. $S_2: (x > 0) \wedge (b \to i \geq 0)$. infinite states!

For state transitions, use two copies of state variables, $R(\vec{x}, \vec{x}')$.

ex) $\Omega_1 = \{p \in \{0,1\}\}$, $R_1: (p \to \neg p') \wedge (\neg p \to p')$. a function!

$\Omega_2 = \{n \in \mathbb{Z}\}$, $R_2: (n' = n+1) \vee (n' = n)$. not a function!

Reachability Analysis: iterate $succ(\phi(\vec{x})) := \{\exists \vec{x}: \phi(\vec{x}) \wedge Trans(\vec{x}, \vec{x}')\}$,

then relabel the set $\{\vec{x}'\} \mapsto \{\vec{x}\}$.

In boolean logic, this is easy since $\exists x: \phi(x) \equiv \phi(x)[x \rightsquigarrow 0] \vee \phi(x)[x \rightsquigarrow 1]$

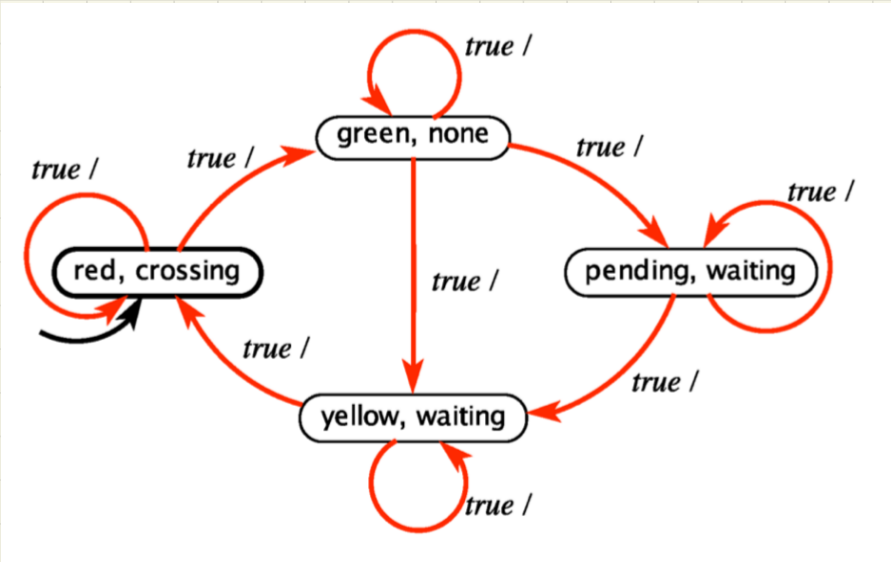
↳ Two useful techniques: Binary Decision Diagram (BDD) / Boolean SAT

→ Still, we might not need to iterate all states.

Abstraction in Model Checking: Use the simplest model that still satisfies

↳ How do we know what to abstract away? → Domain knowledge

↳ One strategy is Localization Abstraction; hide variables not in the property!



Safety vs Liveness Property: Safety only needs a finite-length counterexample; liveness requires infinite-length. [more in LS 15.4]

Controller Synthesis: Negate, then feed into model checker → trace!

Bounded Model Checking (BMC): can a bad state be reachable in n steps?

↳ reducable to SAT with bounded horizon, SMT is "fast enough"

Suppose I have $Init(\vec{x})$, $Trans(\vec{x}, \vec{x}')$, $Bad(\vec{x})$.

In 0 steps: $SAT(Init(\vec{x}) \wedge Bad(\vec{x}))$.

In 1 step: $SAT(Init(\vec{x}_0) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge Bad(\vec{x}_1))$

In n steps: $SAT(Init(\vec{x}_0) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Trans(\vec{x}_{k-1}, \vec{x}_k) \wedge Bad(\vec{x}_k))$

↳ Exhaustively search all of these. Sound (reachable!) but not complete.

Discrete Events

Explicit notion of time (global order), events happen asynchronously

Input allows execution that has nonzero execution time

Time determines order of execution, DE simulators hold a queue

↳ we can have simultaneous events, prevent by inserting Δ-delays

↳ for 0-delay, we might need to resort to fixed point analysis

Pure Asynchrony: never simultaneous, bounded (arbitrary) delays

↳ low computation time, but difficult analysis with addition of time!

may be beneficial for systems insensitive to internal delay & external timing

Dataflow Networks: Actors, Tokens, and Firings. Scheduling??

↳ partially ordered model, deterministic execution independent of schedule

Ideally, DFN actors have unbounded (↳ really...?) FIFO queues ⇒ computation ∥ communication

Determinacy? Blocking Read is one sufficient condition.

Intitively, actors consume input tokens and produce output tokens FIFO.

Formally, actors operate from sequence of input tokens to sequence of output tokens.

Sequence of tokens $X := [x_1, x_2, \ldots]$. Over execution, queue grows tokens.

Actors are interpreted as functions from sequences to sequences.

Ordering of sequences is a prefix order, which is a partial order.

A chain of sequences is a linear ordering of pairwise comparable elements.

↳ ex) $S = \{[x_1], [x_1, x_2], [x_1, x_2, x_3], \ldots\}$

Process $F: S^p \to S^q$. $Y = (y_1, \ldots, y_p) \leq Y' = (y_1', \ldots, y_p')$ iff $y_i \leq y_i'$ $\forall i \leq p$.

Given a chain $C \in S^p$, $F(C)$ might or might not be a chain

↳ We are interested in ones that do.

Continuity: $\forall C$, $\text{lub}(F(C))$ exists, and $F(\text{lub}(C)) = \text{lub}(F(C))$.

Monotonicity: $\forall$ pairs X, X', $X \leq X' \Rightarrow F(X) \leq F(X')$.

A network is a mapping F, $X = F(X, I)$ (↳ inputs)

Behavior of network is defined as the unique least fixed point of $X = F(X, I)$.

Kleene's LFP: if F is continuous, then $LFP = LUB(\{F^n(\perp, I) \mid n > 0\})$.

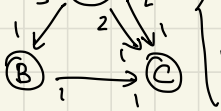
Each process becomes an actor with a firing rule and a function.

Mutually exclusive firing rules $\Rightarrow$ monotinicity! Blocking read is sufficient.

Static DF allows static scheduling during compile time.

↳ a valid schedule must be admissible & periodic.

Balance Equations: $A \xrightarrow{n_p} B$ $V_s :=$ # of firings of actor. $V_s(A) \cdot n_p = V_s(B) \cdot n_c$!

ex) 

$$\begin{cases} 3V_s(A) - V_s(B) = 0, & 2V_s(A) - V_s(C) = 0 \\ V_s(B) - V_s(C) = 0, & 2V_s(A) - V_s(C) = 0 \end{cases} \Rightarrow M\vec{V_s} = 0$$

Full rank $M \Rightarrow$ no non-zero solution! (unique solution is to not fire)

Non-full rank $M \Rightarrow$ infinite solution, but choose the minimum integer solution

In fact, a connected SDF must have at least $(n-1)$ edges, so it makes sense that $rank(M) = n$ or $(n-1)$!!! [Lee '86]

Admissibilty: deadlock at start? add some initial states... maybe.

↳ changes to initial state might impact the functionality! (FIR)

↳ repeatedly schedule fireable actors upto # in repetition vector.

↳ if deadlock before initial state, no valid schedule exists [Lee '86]

Synthesis? Now multiple considerations: time, memory, code size, buffer?

ex) Code stitching: subroutine calls are expensive, loops are cheap

↳ ABCBC ⇒ A(2BC), ABBCC ⇒ A(2B)(2C), these are optimal for code size minimization, single appearance schedule!

↳ for buffer size minimization, we can factor out common divisors for adjacent loops! (100A)(100B)(10C)D ⇒ (10(10A)(10B)C)D

reduced buffer size 210→30, but # of loop instantiation 3→21!

Static DF has limited power, no run-time choices! (ex. Gaussian Elim.)

Non-Static DF is kind of too powerful (Turing-Complete)

Boolean DF is semi-static with some patterns: if-else, do-while, etc.

General Case: thread-based dynamic scheduling

↳ finds a solution if schedulable (sound), but not always terminating

*In reality, FSM and DF are often used together complimentarily!

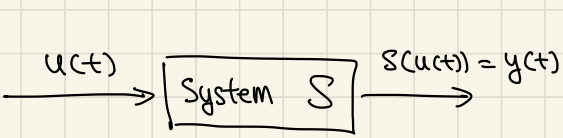
Continuous Systems

Motivation: quantity is explicitly continuous (time, temperature, position), or # of enumerable units is intractable (charges, molecules, population)

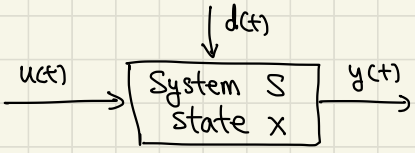
However... computers are discrete, and we need to leverage approximations and numerical schemes that often cannot be implemented exactly

Def) Signal: function $x: \mathbb{T} \to \mathbb{D}$ where $\mathbb{T}$ can be $\mathbb{N}$ or $\mathbb{R}^+$, $\mathbb{D}$ can be $\{0,1\}$, $\mathbb{R}$, or $\mathbb{R}^n$.

Def) System: function $S: (\mathbb{T}_1 \to \mathbb{D}_1) \to (\mathbb{T}_2 \to \mathbb{D}_2)$ that transforms a signal domain to another signal domain.



(SISO system)



(with states)

We can get a differential equation $\dot{x}(t) = f(x(t))$, but how to compute??

Initial Value Problem: show that (under mild assumptions on f) the system $\begin{cases} \dot{x}(0) = x_0 \\ \dot{x}(t) = f(x(t)) \end{cases} \longrightarrow x(t)$ produces a unique solution $x(t)$.

Forward Euler Method: $\tilde{x}(0) = x_0$, $\tilde{x}(t+\delta t) = \tilde{x}(t) + f(t, \tilde{x}(t))\,dt$

↳ How to quantify the error, $Err(t, dt) = \|\tilde{x}(t) - x(t)\|^2$?

An analytical solution is always possible: $x(t) = x_0 + \int_0^t f(x(\tau))\,d\tau$.

↳ Not really helpful, but what if we have $x(t) \mapsto T[x](t)$ on LHS?

$\Rightarrow$ Fixed Point!!

Thm) If f is continuous and Lipschitz, then IVP has a unique solution.

Proof Sketch: By Picard's Iteration, $X_{n+1}(t) = T[X_n](t) = X_0 + \int_0^t f(X_n(\tau))d\tau$ we show that the operator is contracting and thus has a unique FP.

ex) $\dot{X} = 2X$, $X_0 = 1$. Apply Picard's, $X_0(t) = 1$, $X_1(t) = 1+2t$, $X_2(t) = 1+2t+\frac{(2t)^2}{2!}$ which converges to $X(t) = e^{2t} = \sum_{k=0}^{\infty} \frac{(2t)^k}{k!}$.

IVP, numerically: find sequence $(t_n, \hat{X}_n)$ s.t. $t_{n+1} = t_n + h_n$, $\hat{X}_n \simeq X(t_n)$.

The one-step error depends on the order of expansion n & step size h.

↳ Roughly, the truncation of HO terms of Taylor expansion. $Err \leq Ch^n$.

However, global stability is also dependent on the nature of the system!

Approximation Errors: HO Taylors could be computationally challenging

↳ HO derivatives of f are approximated by many evaluations of f

↳ Also, roundoff errors might plague HO estimates!

Backward Euler: $\hat{X}(t+h) = X(t) + f(t+h, X(t+h))h$

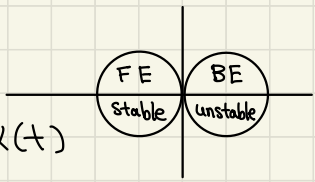
Linear Systems: $\dot{x} = Ax$, $x(t=0) = x_0$. Analytically, $X(t) = e^{At}x_0$ where e^{At} is a Taylor series $I + At + \frac{t^2}{2!}A^2 + \frac{t^3}{3!}A^3 + \cdots$.

Stability: $Re\{\lambda\} < 0$.

FE: $\hat{x}(k+1) = [I + hA]\hat{x}(k)$. Numerically stable if $\|I + hA\| \leq 1$.

↳ This is true if all eigenvalues of hA live inside unit circle at $(-1, 0)$.

↳ choice of h will affect the stability of FE!



BE: $\hat{x}(t+h) = x(t) + hA\hat{x}(t+h) \rightarrow [I - hA]\hat{x}(t+h) = x(t)$

↳ we want $\|I - hA\|^{-1} \leq 1 \Rightarrow$ everything <u>outside</u> unit circle at $(1, 0)$ is stable!

↳ this is stable, but sometimes too stable that it dampens the behavior

Rule of thumb: use both explicit & implicit methods (FE & BE) for validation

One-step Explicit Schemes: $x_{n+1} = x_n + h_n \Phi(x_n, t_n, h_n)$

Also can do variable-step or Runge-Kutta methods.

Closed system: $\dot{x} = f(t, x) \rightarrow x(t)$

Open Loop: $u(t) \rightarrow \dot{x} = f(t, x, u) \rightarrow x(t)$

Closed Loop: $\dot{x} = f(t, x, u) \rightarrow x(t)$, $u(x(t))$

} Existence? Uniqueness?

Computation of $x(t)$?

↳ Ok if continuous & local Lipschitz

↳ But reality is not that good...

Many real-life inputs are step-functions. Moreover, complex feedback can result in differential algebraic equations (DAEs), not only ODEs!

Nonlinear: $f(x) = 0$? Newton-Raphson Method.