What is Verification?

Oded Maler

CNRS-Verimag, Grenoble

August, 2008

Plan of the Talk

- Context: model-based system design (mathematical models)

Plan of the Talk

- Context: model-based system design (mathematical models)
- Example: how to have your coffee, drink it too (and write a paper about it)

Plan of the Talk

- Context: model-based system design (mathematical models)
- Example: how to have your coffee, drink it too (and write a paper about it)
- Major issues in discrete verification

Plan of the Talk

- Context: model-based system design (mathematical models)
- Example: how to have your coffee, drink it too (and write a paper about it)
- Major issues in discrete verification
- New challenges: Timed and Hybrid systems

Context: Building Systems and Mechansims

- We want to build something (a "system") that works
- The system should achieve some of our goals, it should make parts of our world behave in certain way
- We want to build a "good" system that works, not a bad one that fails.

Context: Building Systems and Mechansims

- We want to build something (a "system") that works
- The system should achieve some of our goals, it should make parts of our world behave in certain way
- We want to build a "good" system that works, not a bad one that fails.
- Examples:
  - a house
  - a micro-processor
  - a web server
  - a political system
  - a railway network
  - a car, an airplane, a ship
  - a mobile phone
  - a football team
  - a chemical plant
  - ...

Major Issues

- **What** we want the system to do?
- How do we express what we want (specify) ?
- What type models to use during design?
- How to design it correctly?
- How to build it physically (move from models to real things)?
- How to check whether it really works?
- How to operate and maintain it?

A Side Remark in Sepcification

- Specification is very important also as part of the legal contracts between the provider of the system, sub-contractors and customers
- How can we claim in “objective” and observable terms that a product does or does not work as it should?

A Side Remark in Sepcification

- Specification is very important also as part of the legal contracts between the provider of the system, sub-contractors and customers
- How can we claim in "objective" and observable terms that a product does or does not work as it should?
- Electronnic components have characteristic curves that specify their I/O response
- My cell phone does not work in submarines!

A Side Remark in Sepcification

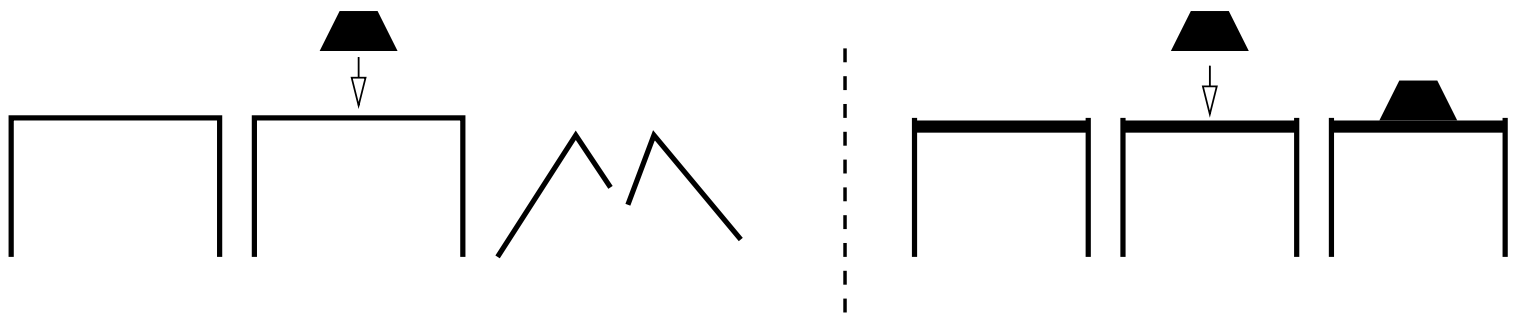
- Specification is very important also as part of the legal contracts between the provider of the system, sub-contractors and customers
- How can we claim in "objective" and observable terms that a product does or does not work as it should?
- Electronnic components have characteristic curves that specify their I/O response
- My cell phone does not work in submarines!
- Of course, there is a limit to formalization and human judges are unavoidable.

Example: Building a House by Trial and Error

- What do we want from a house? Many things (aesthetics, isolation, functioning of sub-systems, ...)
- In particular: we want it not to crash under certain loads

Example: Building a House by Trial and Error

- What do we want from a house? Many things (aesthetics, isolation, functioning of sub-systems, ...)
- In particular: we want it not to crash under certain loads
- An old-fashioned, and still the most popular way to achieve it: build and see (trial and error)



Building a House - using a Model

- Based on physical laws and experiments we can build a **model**, capturing the important features for the problem
- We use the model to predict the behavior (Gedanken experiments).

Building a House - using a Model

- Based on physical laws and experiments we can build a **model**, capturing the important features for the problem
- We use the model to predict the behavior (Gedanken experiments).

- Maximal bending moment on a beam of length l under a load P is $P \cdot l/4$
- Module of resistance of a beam with $b \times h$ section is $b \cdot h^2/6$
- ...

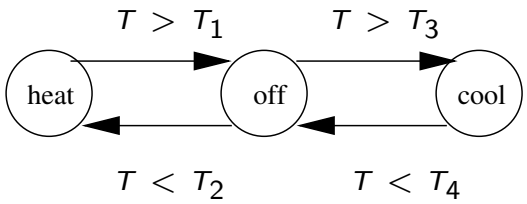
Building a House - using a Model

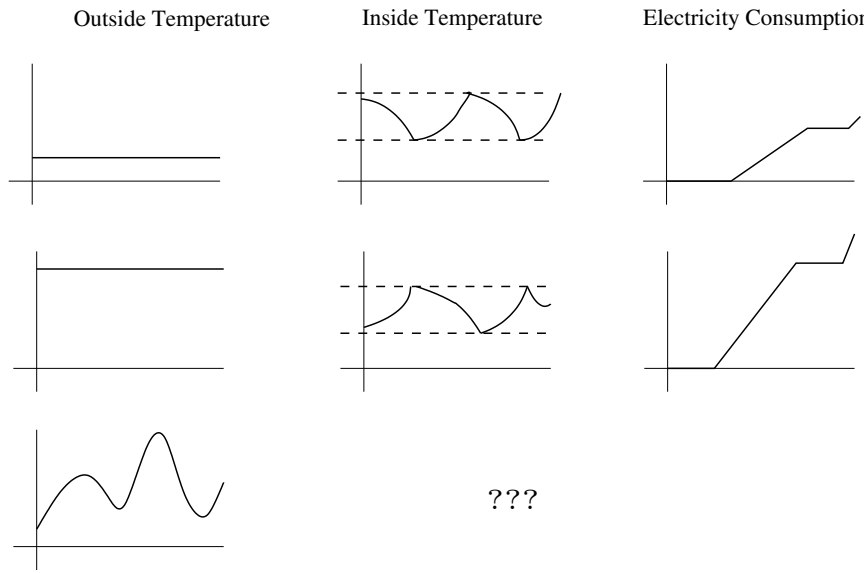
- Based on physical laws and experiments we can build a **model**, capturing the important features for the problem
- We use the model to predict the behavior (Gedanken experiments).

- Maximal bending moment on a beam of length l under a load P is $P \cdot l/4$
- Module of resistance of a beam with $b \times h$ section is $b \cdot h^2/6$
- ...
- Finally we can predict whether or not the beam will support the load.

Example: Air-Conditioning

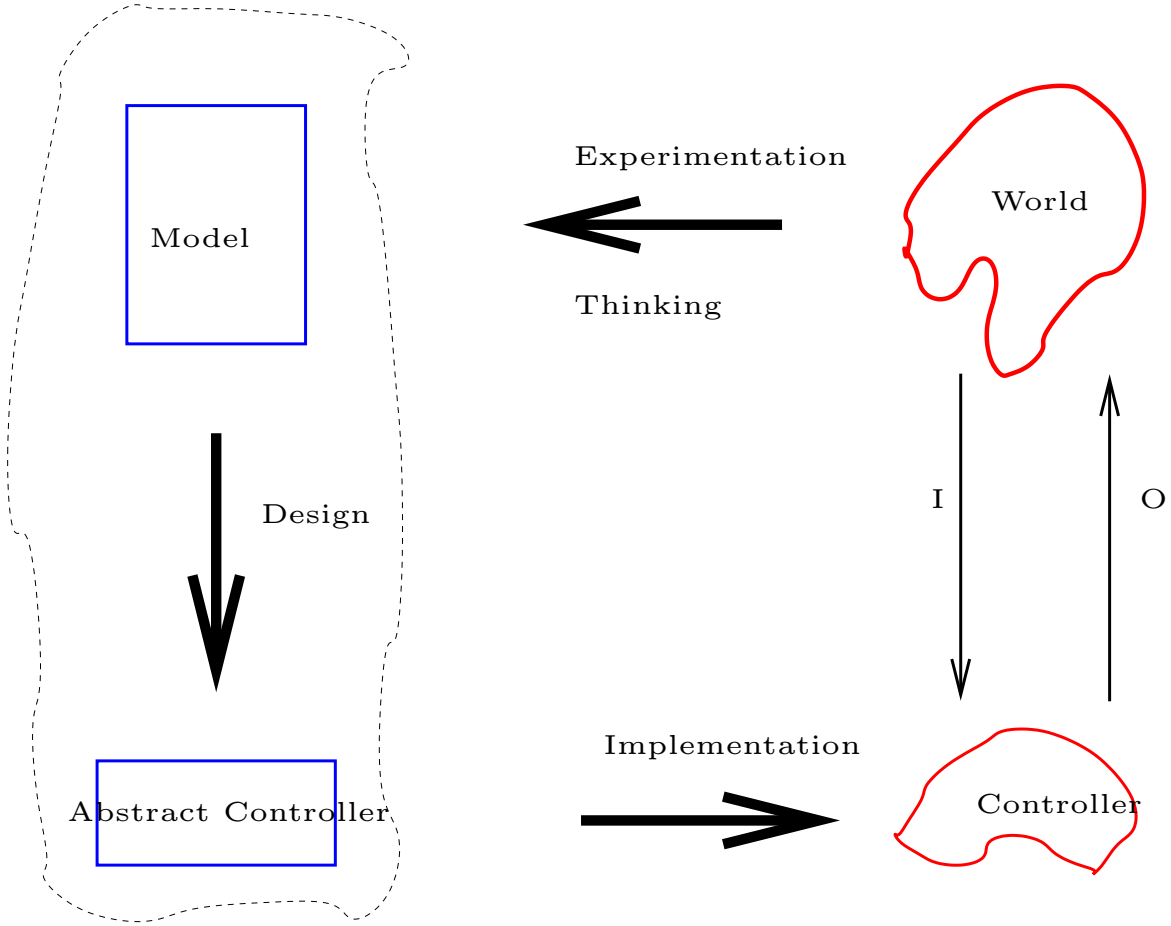
- The house exists and we want to maintain its temperature within certain bounds





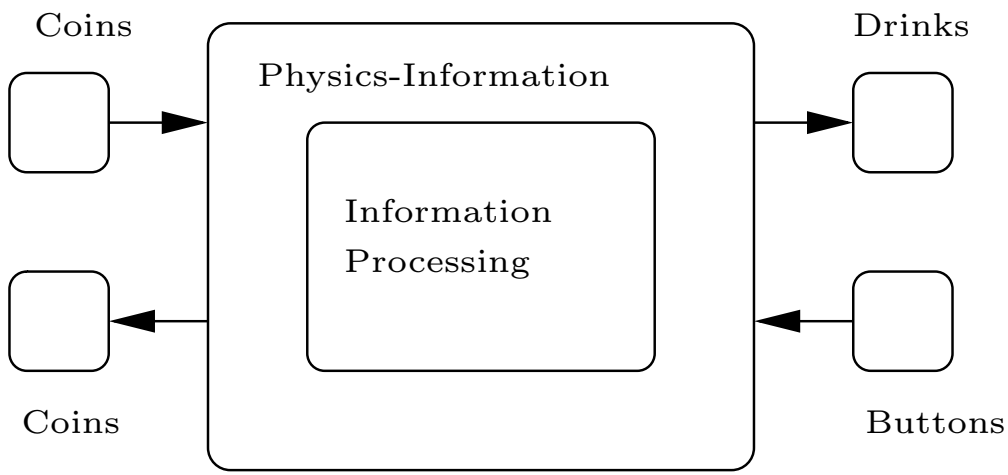
- Can we show that the temperature is **always** maintained in a desired range with some bounded cost?
- Can we show it for **all** external disturbances?

Model-based System Design



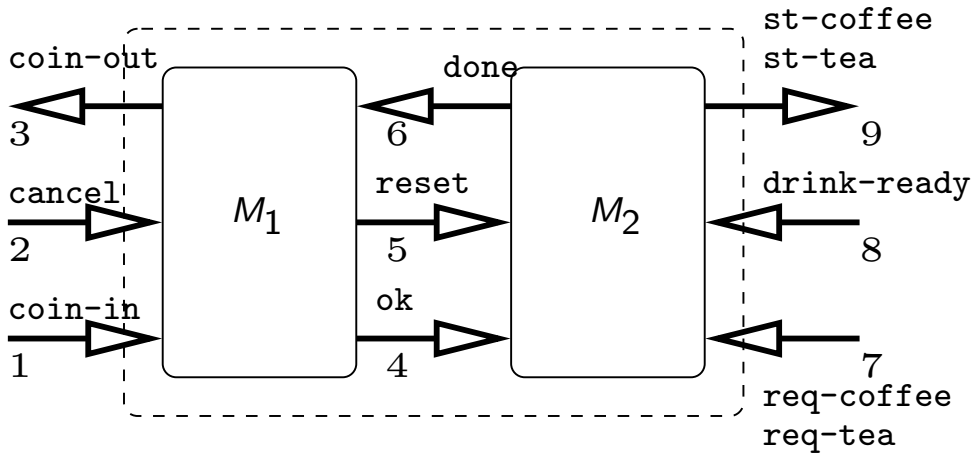
Example: The Coffee Machine

- We want to build a machine that gets coins and delivers coffee or tea
- Modern systems admit a decomposition between physical and information-processing aspects



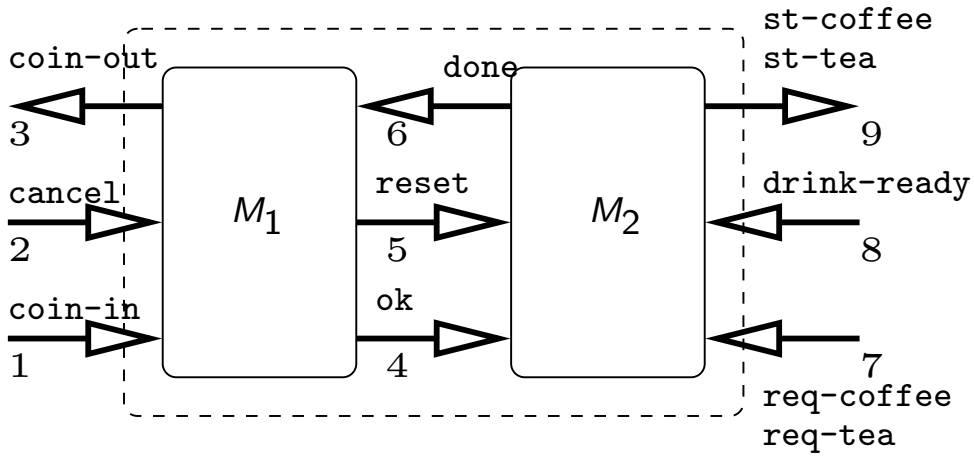
The Coffee Machine: Subsystems and Signals

- We decompse the machine into two interacting subsystems: one for the money, and one for the drinks



The Coffee Machine: Subsystems and Signals

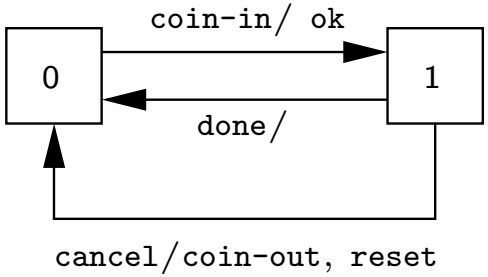
- We decompse the machine into two interacting subsystems: one for the money, and one for the drinks



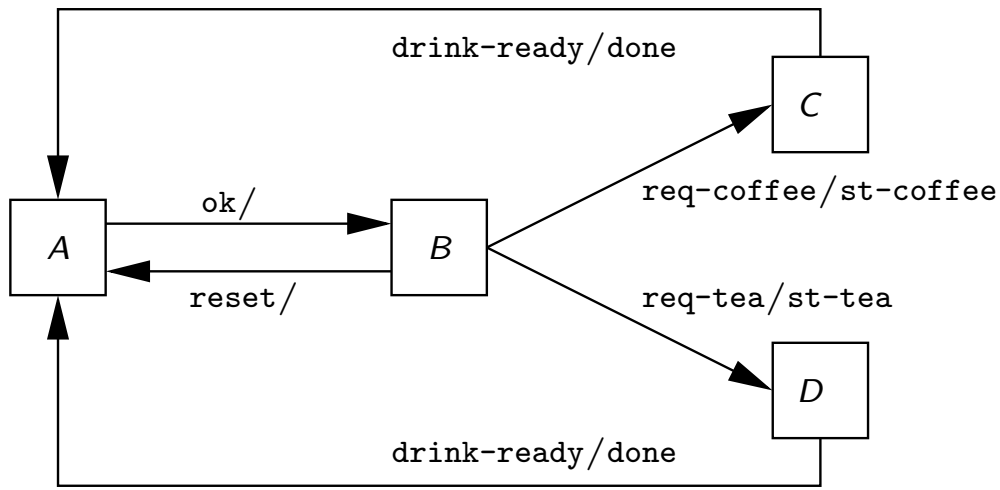
Port	From⟶To	Event types	Meaning
1	$E \longrightarrow M_1$	`coin-in`	a coin was inserted
2	$E \longrightarrow M_1$	`cancel`	cancel button pressed
3	$M_1 \longrightarrow E$	`coin-out`	release the coin
4	$M_1 \longrightarrow M_2$	`ok`	sufficient money inserted
5	$M_1 \longrightarrow M_2$	`reset`	money returned to user
6	$M_2 \longrightarrow M_1$	`done`	drink distribution ended
7	$E \longrightarrow M_2$	`req-coffee` `req-tea`	coffee button pressed tea button pressed
8	$E \longrightarrow M_2$	`drink-ready`	drink preparation ended
9	$M_2 \longrightarrow E$	`st-coffee` `st-tea`	start preparing coffee start preparing tea

The Two Sub-Machines

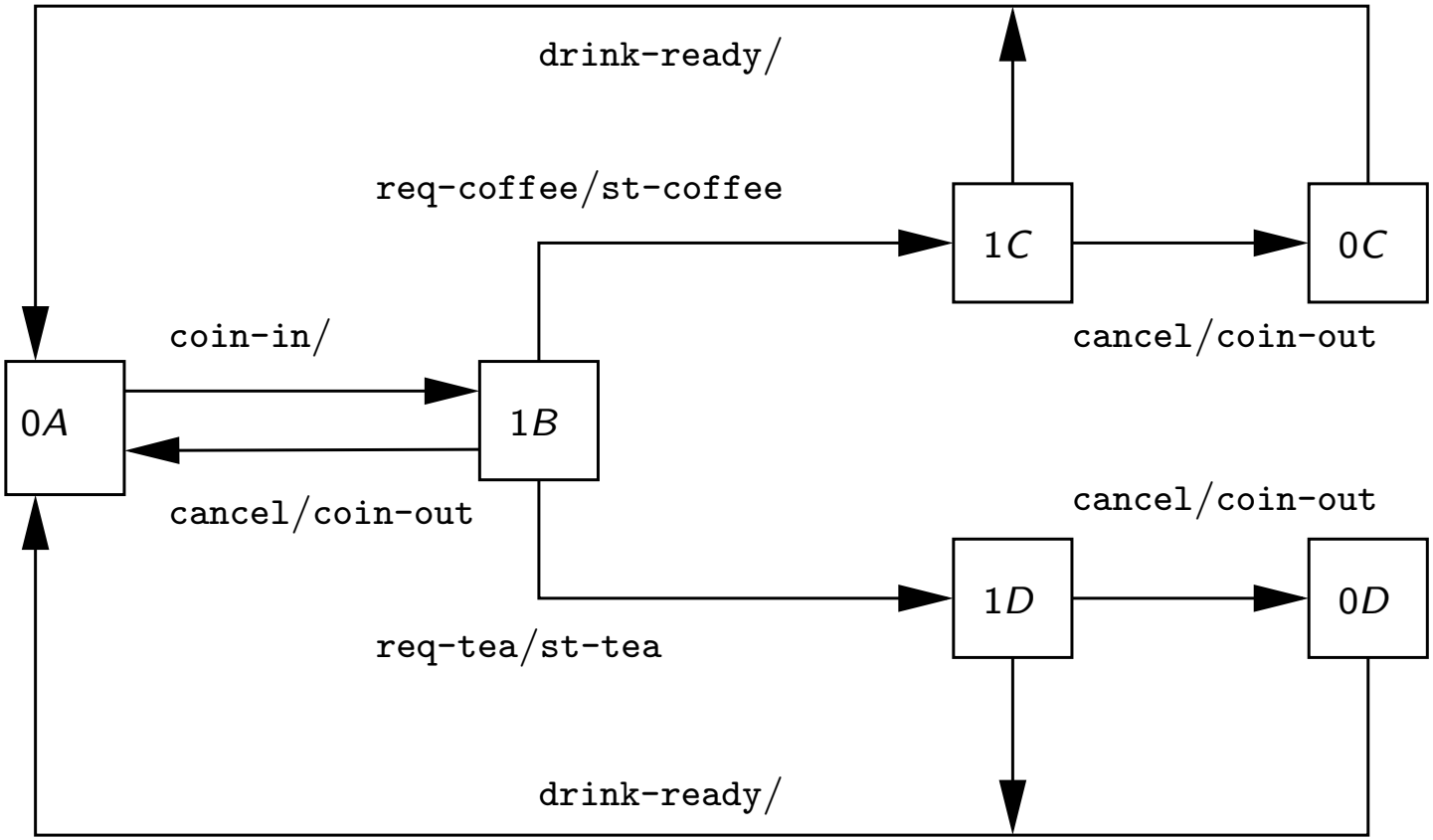
M_1



M_2



The Global Model



Normal behaviors:

$0A$ `coin-in` $1B$ `cancel` `coin-out` $0A$

$0A$ `coin-in` $1B$ `req-coffee` `st-coffee` $1C$ `drink-ready` $0A$

It Could be Much More Complex

- Extensions which require more complex modules:

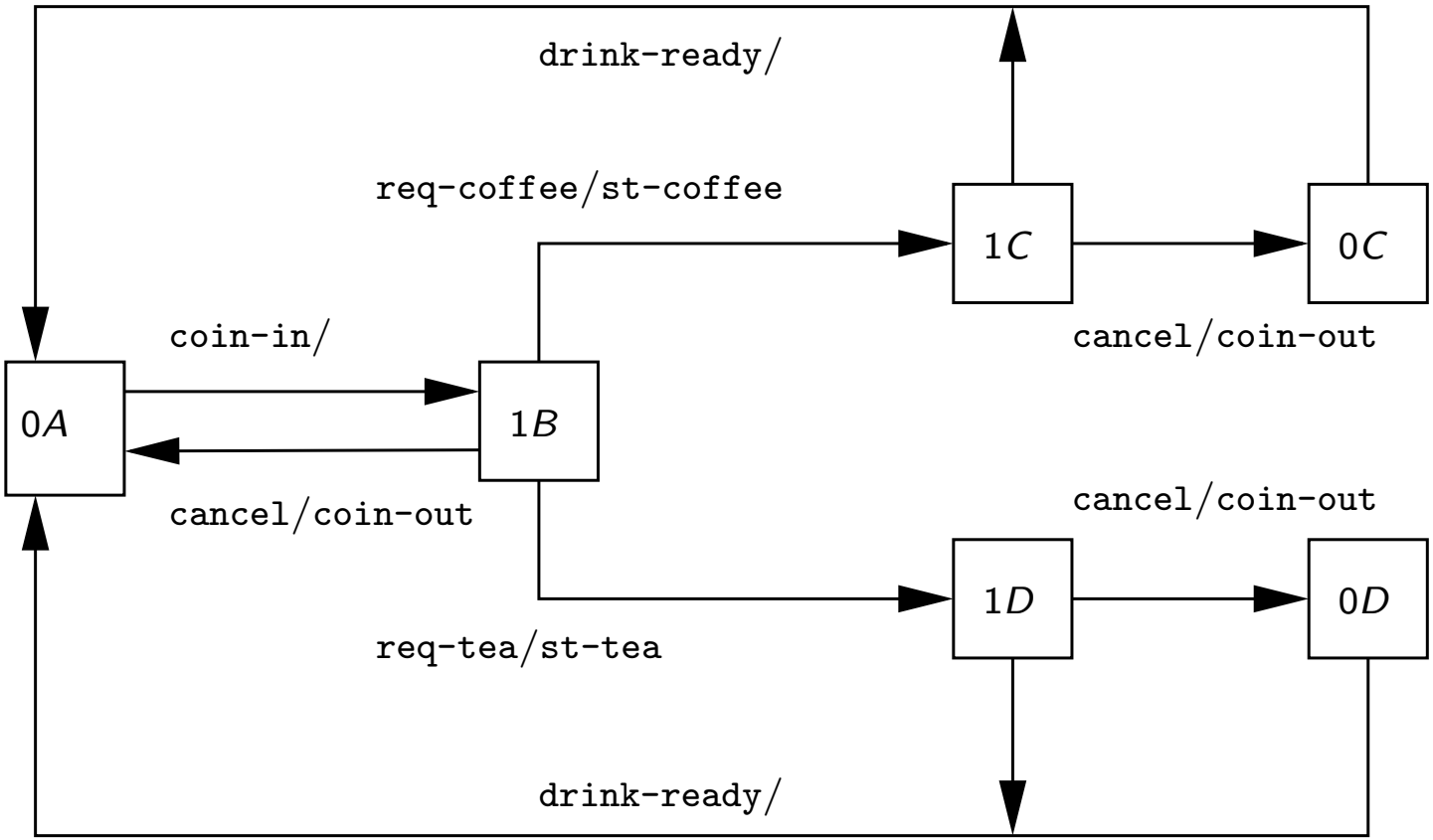
It Could be Much More Complex

- Extensions which require more complex modules:
- Various means of payment: combinations of coins, notes, credit cards (requires a communication module)
- A wider variety of drinks with choices of milk, sugar, etc.
- Consider now a big factory with thousands of components and communication channels

It Could be Much More Complex

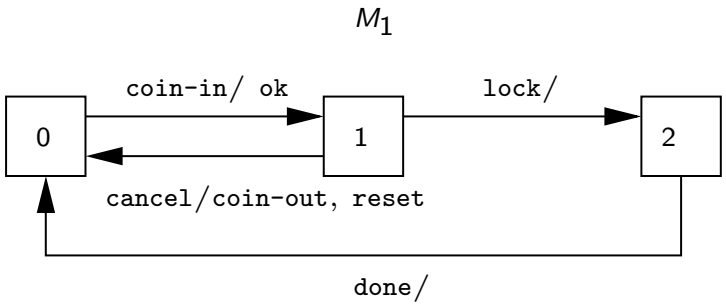
- Extensions which require more complex modules:
- Various means of payment: combinations of coins, notes, credit cards (requires a communication module)
- A wider variety of drinks with choices of milk, sugar, etc.
- Consider now a big factory with thousands of components and communication channels
- When you build a large and complex system with many **interacting** components the number of global states is roughly the **product** of the number of states of the components (**exponential** growth)
- It is practically impossible to predict all the possible behaviors (scenarios) of the system

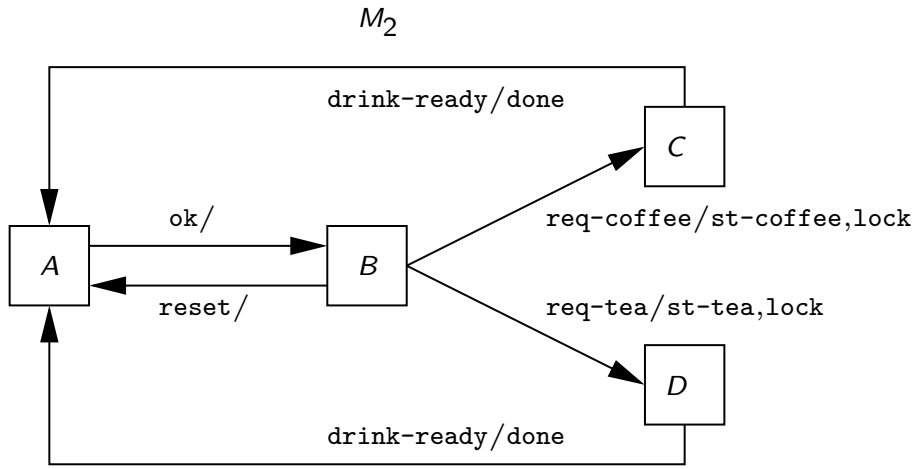
An Unexpected Behavior



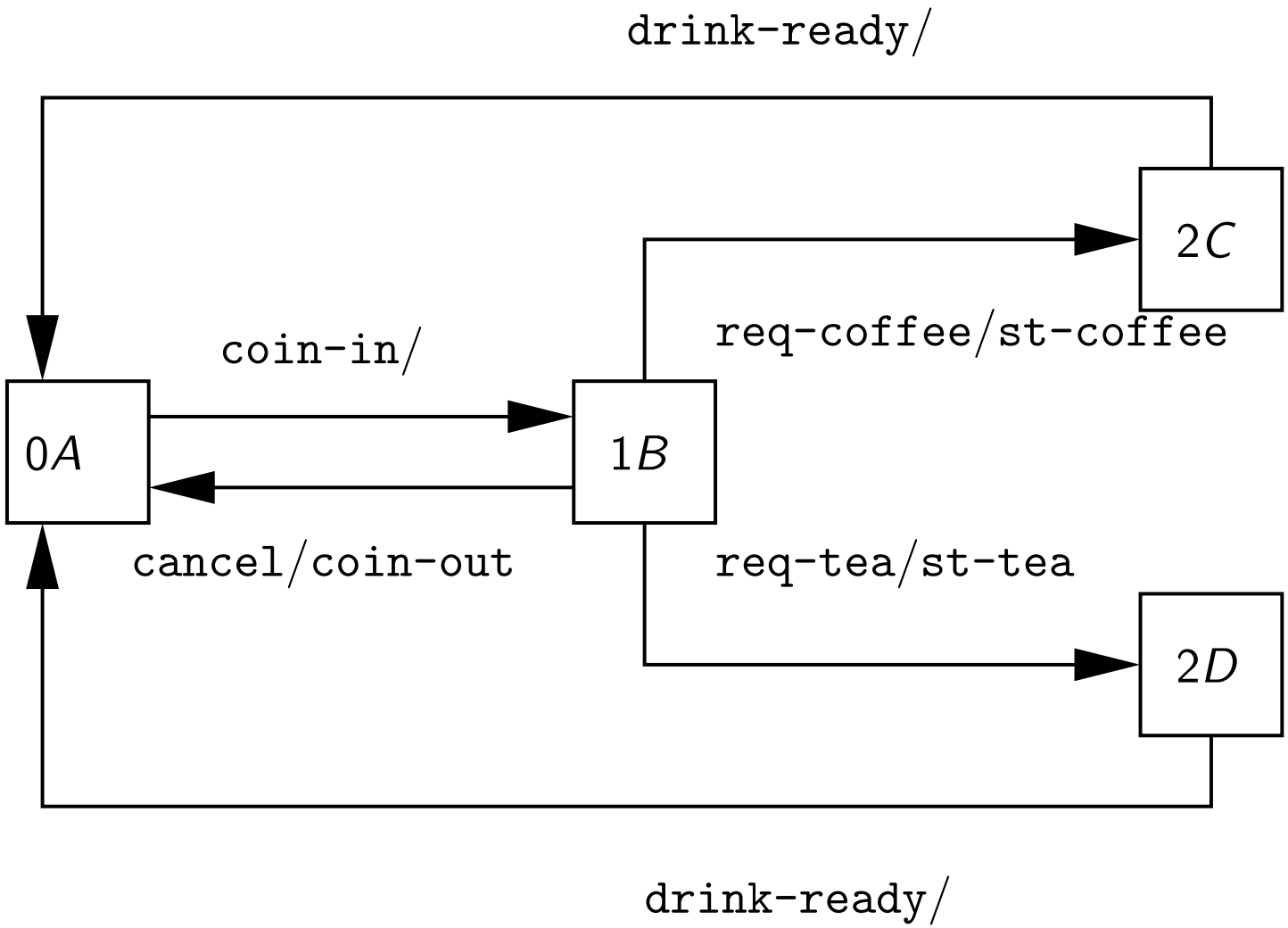
$0A$ coin-in $1B$ req-coffee st-coffee $1C$ cancel coin-out $0C$
drink-ready $0A$

Fixing the Bug





Fixing the Bug – the Global Model



The Moral of the Story I

- Many systems can be modeled as a **composition of interacting automata** (transition systems, discrete event systems).
- Potential behaviors of the system correspond to **paths** in the **global transition graph** of the system.

The Moral of the Story I

- Many systems can be modeled as a **composition of interacting automata** (transition systems, discrete event systems).
- Potential behaviors of the system correspond to **paths** in the **global transition graph** of the system.
- These paths are **labeled** by **input events**, interaction with the environment external to the system
- Each input sequence may induce a **different behavior**
- We want to make sure that a system responds correctly to **all conceivable inputs**

The Moral of the Story II

- For every **individual input sequence** we can **simulate** the reaction of the system
- We cannot do it **exhaustively** due to the prohibitively-huge number of input sequences

The Moral of the Story II

- For every **individual input sequence** we can **simulate** the reaction of the system
- We cannot do it **exhaustively** due to the prohibitively-huge number of input sequences
- Verification is a collection of automatic and semi-automatic methods to analyze **all** the paths in the graph
- This is hard for humans to do and even for computers.

The Ingredients of a Verification Methodology

The Ingredients of a Verification Methodology

- **A Specification Language:**
  A formalism for describing the desired properties of the system. A criterion for classifying event sequences as good and bad (e.g. Temporal Logic)

The Ingredients of a Verification Methodology

- **A Specification Language:**
  A formalism for describing the desired properties of the system. A criterion for classifying event sequences as good and bad (e.g. Temporal Logic)
- **A Computational Model:**
  A formalism for describing the designed system (automata, transition systems, programs)

The Ingredients of a Verification Methodology

- **A Specification Language:**
  A formalism for describing the desired properties of the system. A criterion for classifying event sequences as good and bad (e.g. Temporal Logic)
- **A Computational Model:**
  A formalism for describing the designed system (automata, transition systems, programs)
- **A Verification Technique:**
  A method to show that the system satisfies the desired properties, i.e. all the behaviors generated by the system are those accepted by the specification (deductive and algorithmic approaches).

Specification Languages

- How to specify in a **rigorous** and **precise** manner what the desired properties of the system are
- Temporal Logic is a formalism in which you can express properties of **sequences** of **events**, especially about the order of their occurrences

Specification Languages

- How to specify in a **rigorous** and **precise** manner what the desired properties of the system are
- Temporal Logic is a formalism in which you can express properties of **sequences** of **events**, especially about the order of their occurrences
- If a customer puts the right amount of money and chooses a drink then he will **later** get the chosen drink.
- If a customer selects a drink and the process has started the `cancel` button is ignored
- If the customer has put money and 30 seconds have elapsed before a drink is selected, the money is given back

The Deductive Approach to Verification

- Formalization of human reasoning:

The Deductive Approach to Verification

- Formalization of human reasoning:
- IF `req-coffee` causes a `lock` message from M_1 to M_2 before `st-coffee`
- AND a `lock` message makes M_1 move to state 2
- AND in state 2, M_2 ignores `cancel` messages
- THEN it is impossible to get a free coffee.

The Deductive Approach to Verification

- Formalization of human reasoning:
- IF `req-coffee` causes a `lock` message from M_1 to M_2 before `st-coffee`
- AND a `lock` message makes M_1 move to state 2
- AND in state 2, M_2 ignores `cancel` messages
- THEN it is impossible to get a free coffee.

- In order to show correctness of the system we have to prove many many small and boring theorems.
- Here the computer and the human **cooperate** in the verification process.
- The human (who has **intuition** about the system) suggests proof directions and the computer checks, does the book-keeping, etc.

The Algorithmic Approach to Verification

- Brute-force search:

The Algorithmic Approach to Verification

- Brute-force search:
- Graph algorithms are applied to the global transition graph of the system to detect bad behaviors (or to prove their absence).
- Advantages: you don't need an intelligent user (an endangered species) – **in principle** you just push a button and the computer answers.

The Algorithmic Approach to Verification

- Brute-force search:
- Graph algorithms are applied to the global transition graph of the system to detect bad behaviors (or to prove their absence).
- Advantages: you don't need an intelligent user (an endangered species) – **in principle** you just push a button and the computer answers.
- Problem: **state-explosion** – the number of states can be 2^{100} beyond the capabilities of the fastest (present and future) computers
- Most of the work: inventing tricks to treat larger problems
- E.g. symbolic representation of large graphs, compositional reasoning, approximation and abstraction, combination with deductive methods
- This will be explained the rest of this talk

Model I: Closed Systems

- A transition system is $S = (X, \delta)$ where X is finite and $\delta : X \to X$ is the transition function.
- The state-space X has no numerical meaning and no interesting structure (unlike numbers in $\mathbb{N}$ or $\mathbb{R}$)

Model I: Closed Systems

- A transition system is $S = (X, \delta)$ where X is finite and $\delta : X \to X$ is the transition function.
- The state-space X has no numerical meaning and no interesting structure (unlike numbers in $\mathbb{N}$ or $\mathbb{R}$)
- Notation: X^k is the set of all sequences of length k; X^* the set of all sequences.
- **Behavior:** the behavior of S starting from an initial state $x_0 \in X$, is

$$\xi = \xi[0], \xi[1], \ldots \in X^*$$

s.t. $\xi[0] = x_0$ and for every i, $\quad \xi[i+1] = \delta(\xi[i])$

Model I: Closed Systems

- A transition system is $S = (X, \delta)$ where X is finite and $\delta : X \to X$ is the transition function.
- The state-space X has no numerical meaning and no interesting structure (unlike numbers in $\mathbb{N}$ or $\mathbb{R}$)
- Notation: X^k is the set of all sequences of length k; X^* the set of all sequences.
- **Behavior:** the behavior of S starting from an initial state $x_0 \in X$, is

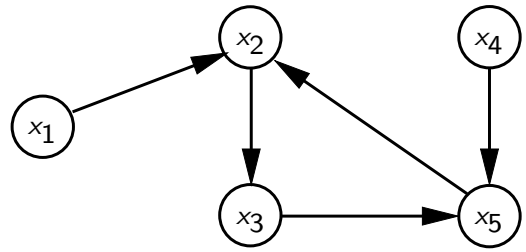
$$\xi = \xi[0], \xi[1], \ldots \in X^*$$

  s.t. $\xi[0] = x_0$ and for every i, $\quad \xi[i+1] = \delta(\xi[i])$

- Basic Reachability Problem: Given x_0 and a set $P \subseteq X$, does the behavior of S starting at x_0 reach P?

Solution by Forward Simulation

$\xi[0] := x_0$
$F^0 := \{x_0\}$
repeat
  $\xi[k+1] := \delta(\xi[k])$
  $F^{k+1} := F^k \cup \{\xi[i+1]\}$
until $F^{k+1} = F^k$
$F_* := F^k$



$$\{x_1\}, \{x_1, x_2\}, \{x_1, x_2, x_3\}, \{x_1, x_2, x_3, x_5\}$$

- How to do it for continuous system defined by $\dot{x} = f(x)$?

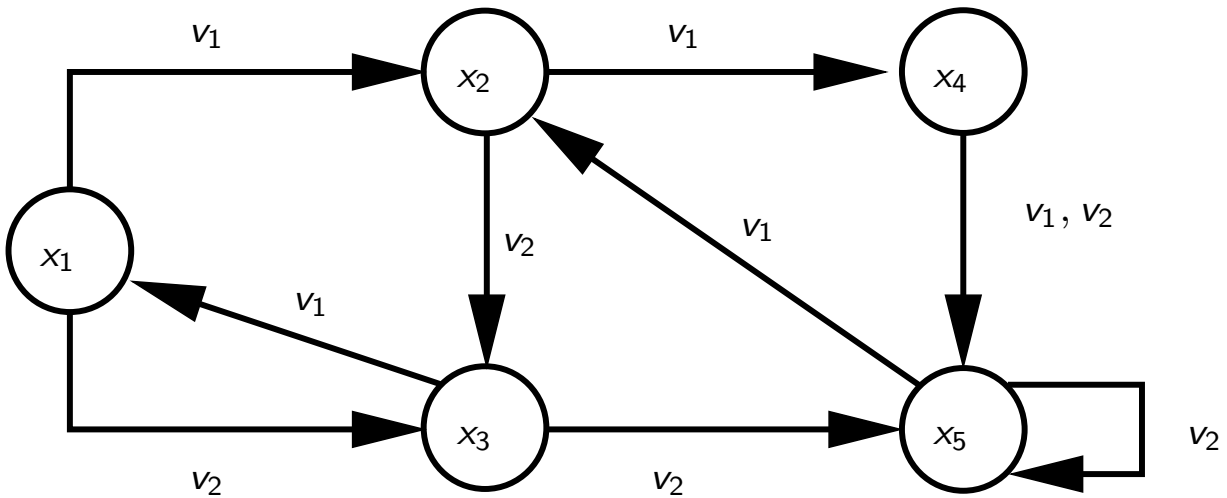
Model II: Systems with One Input

- A one-input transition system is $S = (X, V, \delta)$ where X and V are finite and $\delta : X \times V \longrightarrow X$ is the transition function.

Model II: Systems with One Input

- A one-input transition system is $S = (X, V, \delta)$ where X and V are finite and $\delta : X \times V \to X$ is the transition function.
- Behavior induced by input: Given an input sequence $\psi \in V^*$, the behavior of S starting from $x_0 \in X$ in the presence of ψ is a sequence

$$\xi(\psi) = \xi[0], \xi[1], \ldots \in X^* \text{ such that } \quad \xi[i+1] = \delta(\xi[i], \psi[i]).$$



$$x_1 \xrightarrow{v_1} x_2 \xrightarrow{v_2} x_3 \xrightarrow{v_2} x_5 \xrightarrow{v_1} x_2 \xrightarrow{v_1} x_4$$

Reachability for Open Systems

- The reachability problem: is there **some** some input sequence $\psi \in V^*$ such that $\xi(\psi)$ reaches P?

Reachability for Open Systems

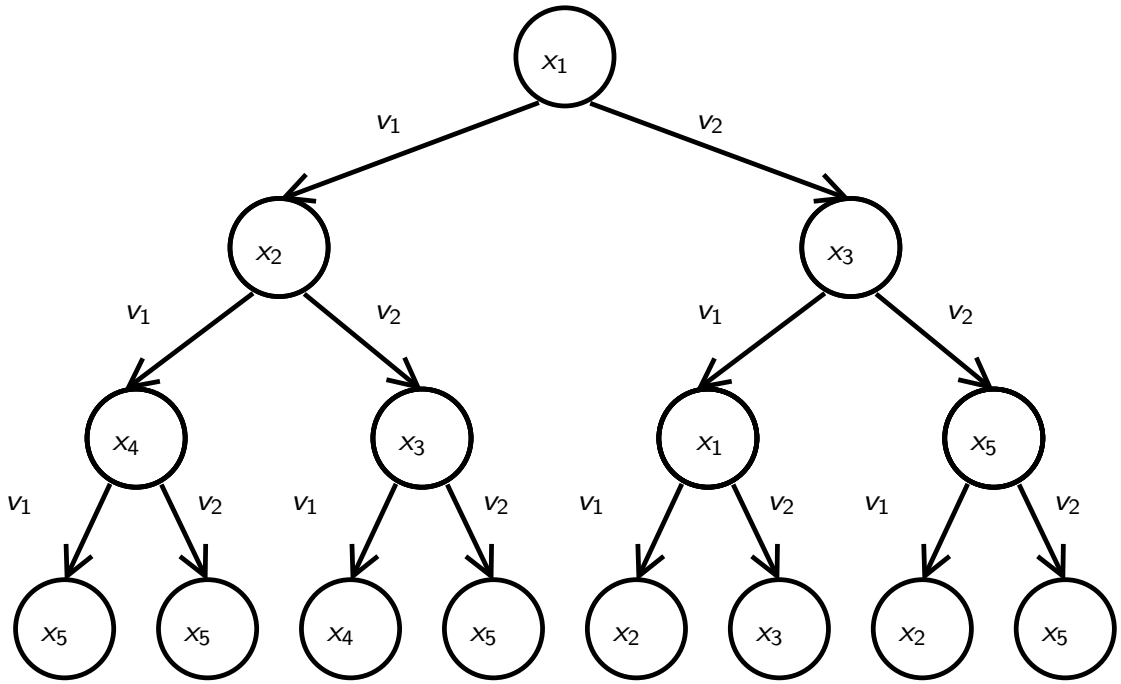
- The reachability problem: is there **some** some input sequence $\psi \in V^*$ such that $\xi(\psi)$ reaches P?
- For every **given** ψ we can use the previous algorithm, simulate and obtain $F_*(\psi)$.

Reachability for Open Systems

- The reachability problem: is there **some** some input sequence $\psi \in V^*$ such that $\xi(\psi)$ reaches P?
- For every **given** ψ we can use the previous algorithm, simulate and obtain $F_*(\psi)$.
- For an automaton with n states all states are reachable by sequences of length $< n$.

$$F_* = \bigcup_{\psi \in V^n} F_*(\psi)$$

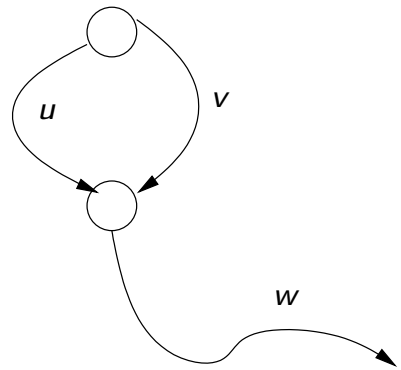
Reachability for Open Systems



- There are 2^n input sequences to simulate with (and n itself is, typically exponential in the number of system components)

A More Efficient Way

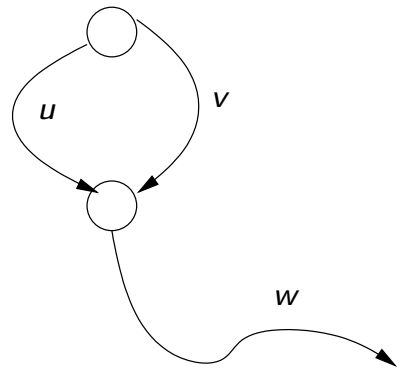
- Many different input sequences lead to the same state and if $\delta(x, u) = \delta(x, v)$ then for every w, $\delta(x, uw) = \delta(x, vw)$.



- We do not need to "simulate" with both uw and vw.

A More Efficient Way

- Many different input sequences lead to the same state and if $\delta(x, u) = \delta(x, v)$ then for every w, $\delta(x, uw) = \delta(x, vw)$.



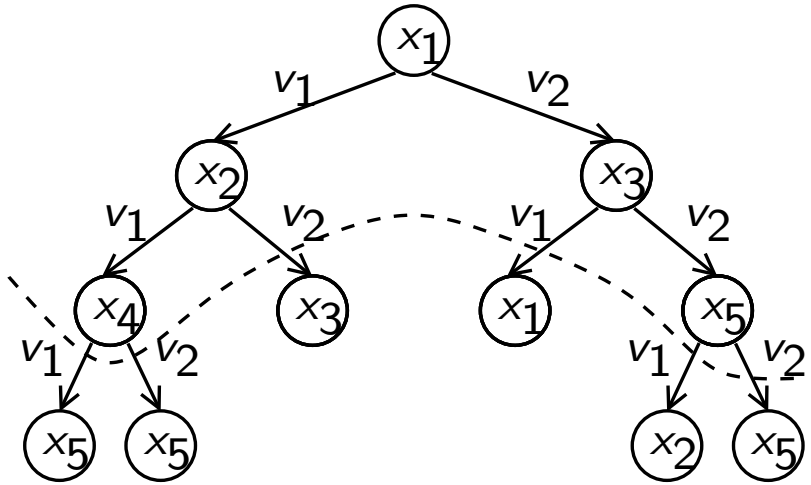
- We do not need to "simulate" with both uw and vw.
- Since we have access to the transition graph (unlike black box) we can apply graph algorithms.
- Immediate successors of a state x:

$$\delta(x) = \{x' : \exists u \ \delta(x, u) = x'\}$$

- Successors of a set F: $\delta(F) = \{\delta(x) : x \in F\}$

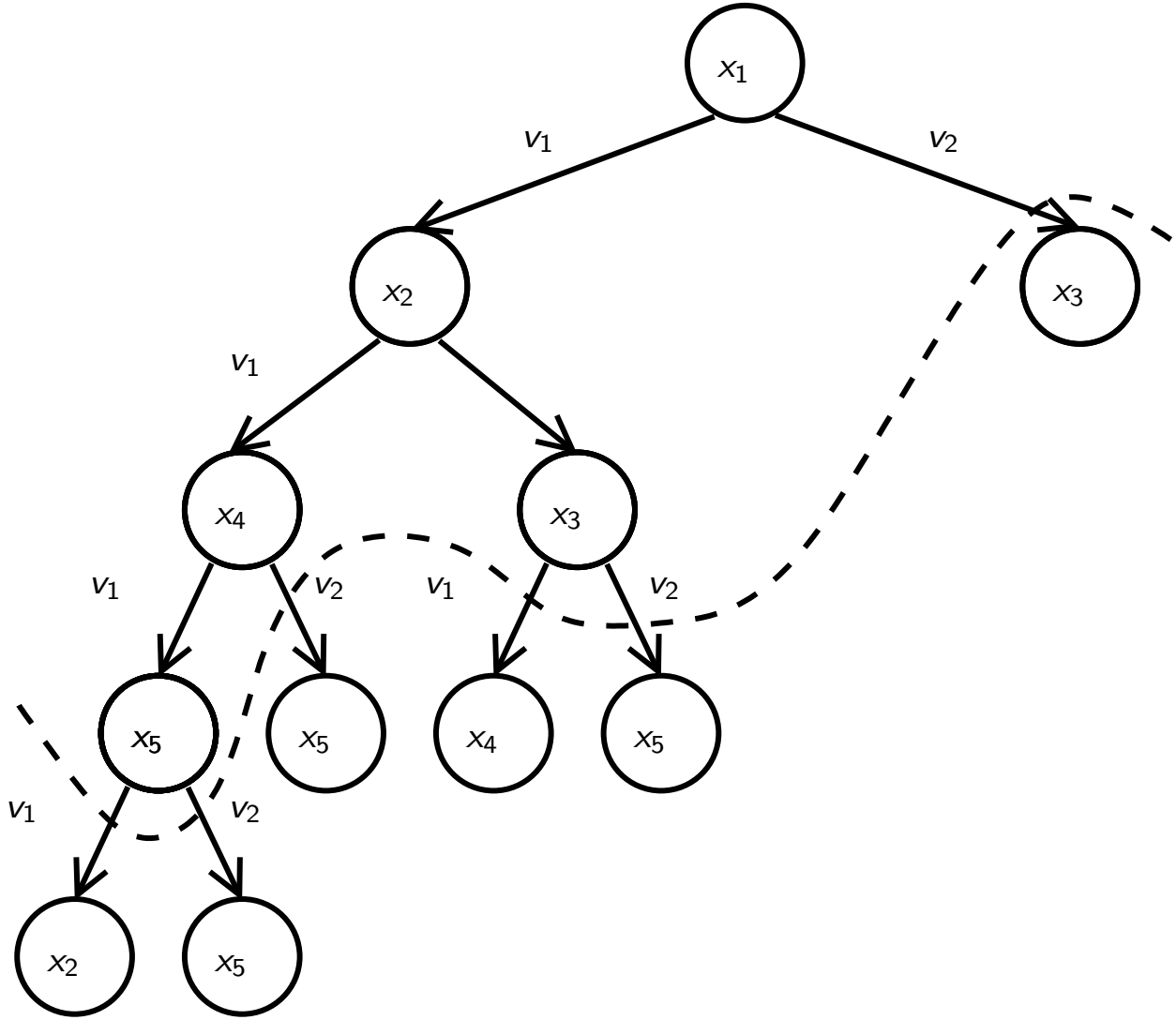
Forward Reachability (breadth-first)

$F^0 := \{x_0\}$
repeat
$F^{k+1} := F^k \cup \delta(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$



- $F_0 = \{x_1\}$,
- $F_1 = \{x_1, x_2, x_3\}$
- $F_2 = \{x_1, x_2, x_3, x_4, x_5\} = F_*$
-
- Complexity: only $O(n \cdot \log n \cdot |V|)$

Variation: Depth-First



Variation: Backwards

- Backwards: find all states from which there is an input that induces a behavior leading to P
- Immediate predecessors of a state x:
- $\delta^{-1}(x) = \{x' : \exists u \ \delta(x', u) = x\}$

$F^0 := P$
repeat
  $F^{k+1} := F^k \cup \delta^{-1}(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$

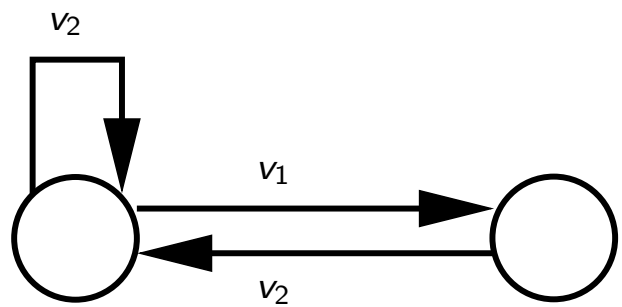
- When P is a set of bad states, a bad behavior is found if F_* contains initial states

Admissible Inputs

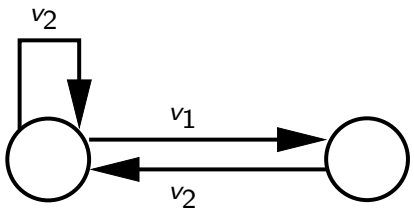
- So far we have assumed that the external environment can generate all sequences in V^*.
- This is as if we modeled the environment as a one-state automaton (the universal generator).

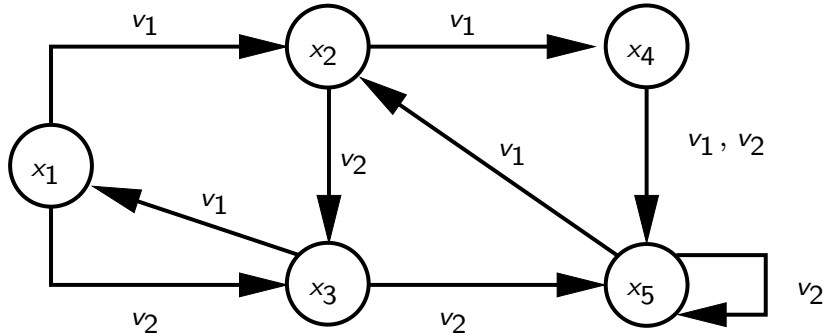
Admissible Inputs

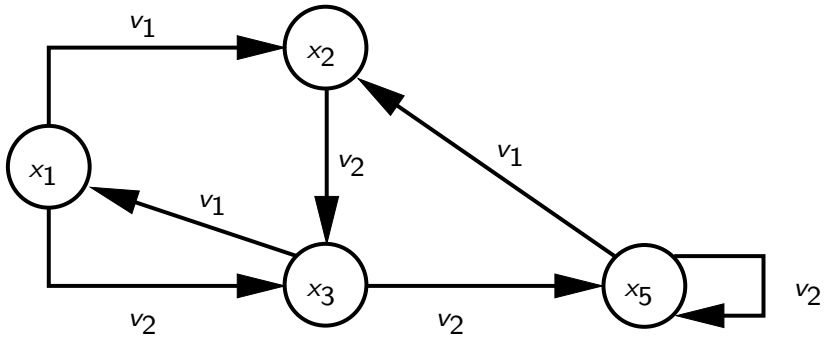
- So far we have assumed that the external environment can generate all sequences in V^*.
- This is as if we modeled the environment as a one-state automaton (the universal generator).
- We can have a more restricted environment, e.g. it will never produce $v_1 v_1$.
- We can build an automaton which models the environment and compose it with the model of the system.



Admissible Inputs - the Composition







Verification: The State-of-the-Art

- Algorithms that take a description of **any** open system and verify whether any of the admissible inputs drives the system into a set P

Verification: The State-of-the-Art

- Algorithms that take a description of **any** open system and verify whether any of the admissible inputs drives the system into a set P
- For more complex temporal properties one can build an automaton for the property and compose it with the system

Verification: The State-of-the-Art

- Algorithms that take a description of **any** open system and verify whether any of the admissible inputs drives the system into a set P
- For more complex temporal properties one can build an automaton for the property and compose it with the system
- Such algorithms are guaranteed to terminate after a finite number of steps
- This is essentially what algorithmic verification ("model checking") is all about.
- This is **general**: it is valid for every discrete finite-state system.
- Of course, finite systems can be **very large** and so the time until termination...

Verification: The State-of-the-Art

- Algorithms that take a description of **any** open system and verify whether any of the admissible inputs drives the system into a set P
- For more complex temporal properties one can build an automaton for the property and compose it with the system
- Such algorithms are guaranteed to terminate after a finite number of steps
- This is essentially what algorithmic verification ("model checking") is all about.
- This is **general**: it is valid for every discrete finite-state system.
- Of course, finite systems can be **very large** and so the time until termination...
- The analogue for continuous systems: do the same for a system defined by $\dot{x} = f(x, u)$.

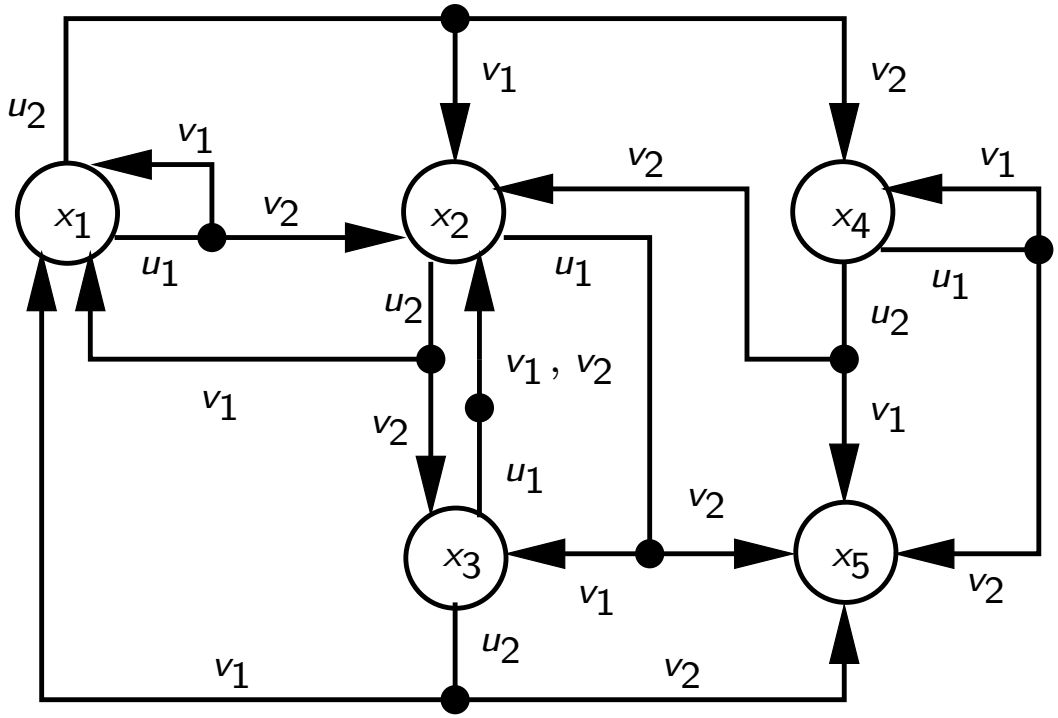
Systems with two Inputs

- A two-input transition system is $S = (X, U, V, \delta)$ where X, U and V are finite sets
- The transition function $\delta : X \times U \times V \rightarrow X$ determines the next state depending on both u and v

Systems with two Inputs

- A two-input transition system is $S = (X, U, V, \delta)$ where X, U and V are finite sets
- The transition function $\delta : X \times U \times V \to X$ determines the next state depending on both u and v
- Interpretation of inputs:
- U: **we**, the good guys, the controller
- V: **they**, the bad guys, disturbances
- An antagonist game situation:
- Our goal is to choose each time an element of U such that all the behaviors induced by all possible disturbances are good

Systems with two Inputs: Example



$$\delta(x_1, u_1, v_1) = x_1 \qquad \delta(x_1, u_1, v_2) = x_2$$

$$\delta(x_1, u_2, v_1) = x_2 \qquad \delta(x_1, u_2, v_2) = x_4$$

Strategies

- **Strategy**: a function $c : X^* \to U$ what to do at each point in the game
- **State strategy**: a function $c : X \to U$ a strategy that depends only on the current state (not history)

Strategies

- **Strategy**: a function $c : X^* \to U$ what to do at each point in the game
- **State strategy**: a function $c : X \to U$ a strategy that depends only on the current state (not history)
- A strategy c converts a type III system into a type II system
- $S_c = (X, V, \delta_c)$ s.t. $\delta_c(x, v) = \delta(x, c(x), v)$.
- **Controller synthesis** problem: Given $S = (X, U, V, \delta)$ and a set $P \subseteq X$ of "bad" states:
- Find a strategy c such that all the behaviors of the derived closed-loop system $S_c = (X, V, \delta_c)$ never reach P

Controllable Predecessors

- The computation of strategies and "winning" state can be computed by a variant of backward reachability using s specialized operator
- For $S = (X, U, V, \delta)$ and $F \subseteq X$, the set of controllable predecessors of F is

$$\pi(F) = \{x : \exists u \in U \ \forall v \in V \ \ \delta(x, u, v) \in F\}$$

- The states from which the controller, by properly selecting u, can force the system into P in the next step.

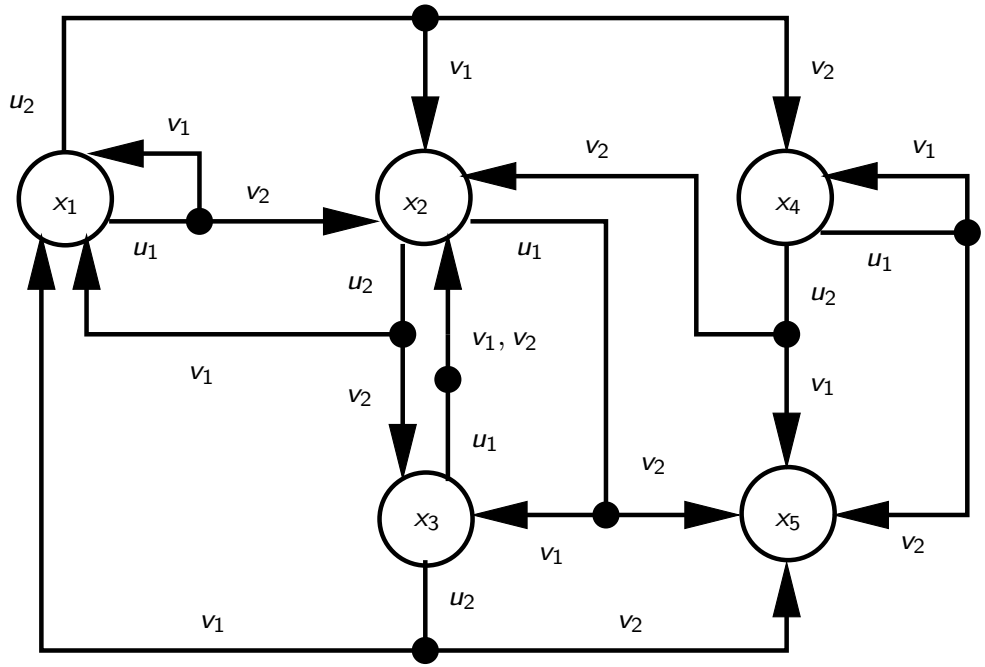
Finding Winning States and Strategies

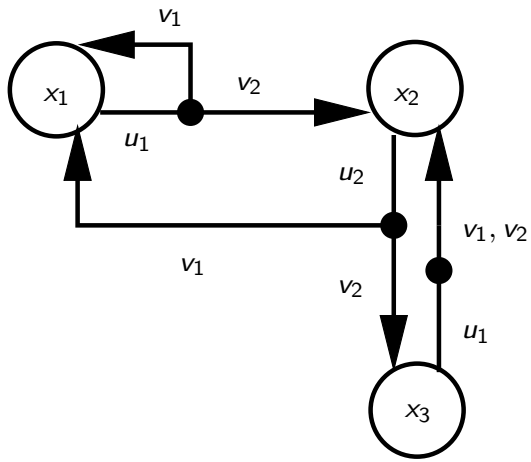
- The following backward algorithm finds the set F_* of "winning states" from which P can be avoided forever.

$F^0 := X - P$
repeat
  $F^{k+1} := F^k \cap \pi(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$

- Remark: this is similar to the Ramadge-Wonham theory of discrete event control, dynamic programming, min-max, game algorithms, MDPs, etc.

Synthesis Example





- We want to avoid x_5

$$F^0 = \{x_1, x_2, x_3, x_4\}, \qquad F^1 = \{x_1, x_2, x_3\} = F_*$$

- The resulting "closed-loop" system always remains in $\{x_1, x_2, x_3\}$.

Remark: Quality vs. Quantity

- Correctness is a special case of the more general notion of a **performance measure**:
- An assignment of a value to each behavior as indication of its goodness

Remark: Quality vs. Quantity

- Correctness is a special case of the more general notion of a **performance measure**:
- An assignment of a value to each behavior as indication of its goodness
- Traditionally verification is concerned with a $\{0, 1\}$ measure (correct or not) and in worst-case reasoning (one bad behavior renders the system incorrect)
- One can assign to system behaviors **numbers** that indicate their cost or utility and then try to synthesize **optimal** controllers

Remark: Quality vs. Quantity

- Correctness is a special case of the more general notion of a **performance measure**:
- An assignment of a value to each behavior as indication of its goodness
- Traditionally verification is concerned with a $\{0, 1\}$ measure (correct or not) and in worst-case reasoning (one bad behavior renders the system incorrect)
- One can assign to system behaviors **numbers** that indicate their cost or utility and then try to synthesize **optimal** controllers
- See my pamphlet **On control in the presence of adversaries**

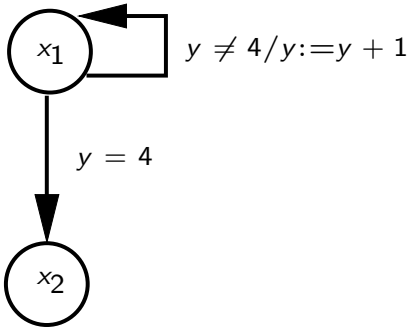
Discrete Infinite-State Systems

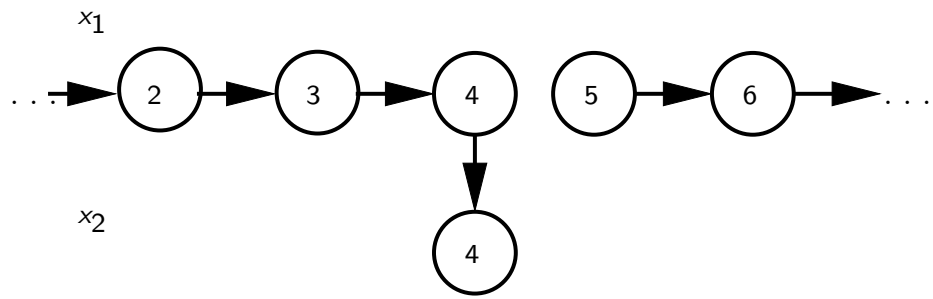
- So far we have dealt with finite-state systems ("control" but no "data").
- Computer programs can be viewed as syntactic representations of discrete dynamical systems with an infinite state-space.

repeat
$y := y + 1$
until $y = 4$

- State space is the product of the set of program locations and the domains of the variables: $\{x_1, x_2\} \times \mathbb{Z}$

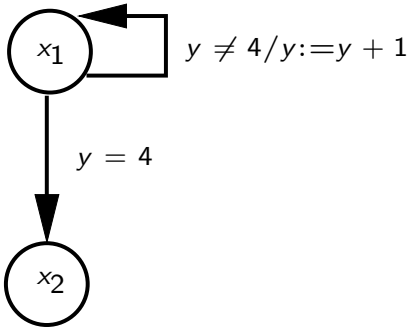
Verification of Infinite-State Systems

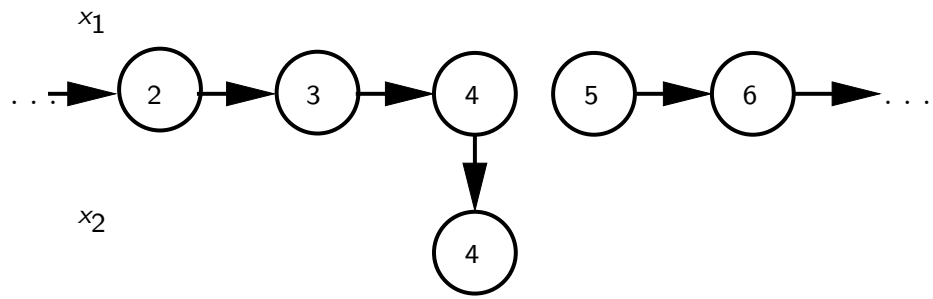




- Forward reachability algorithm will terminate if started from $(x_1, 2)$ but not from $(x_1, 5)$.
- The reachability problem is unsolvable: there is no **general** algorithm that solves every instance of it.

Verification of Infinite-State Systems





- Forward reachability algorithm will terminate if started from $(x_1, 2)$ but not from $(x_1, 5)$.
- The reachability problem is unsolvable: there is no **general** algorithm that solves every instance of it.
- "Deductive" approach: prove properties "analytically", find loop invariants, ...
- "Symbolic" approach: reachability using formulae to represent sets of states, e.g. $x = x_1 \wedge y \geq 5$ or abstract domains

Hybrid Systems: Modeling the Physical Environment

- Most systems are **embedded** in the physical environemnt via sensors and actuators

Hybrid Systems: Modeling the Physical Environment

- Most systems are **embedded** in the physical environemnt via sensors and actuators
- Sometimes it is sufficient to abstract the dynamics of the environment using discrete events (the physical part of the coffee machine emits `drink-ready` sometime after receiving `st-coffee`).

Hybrid Systems: Modeling the Physical Environment

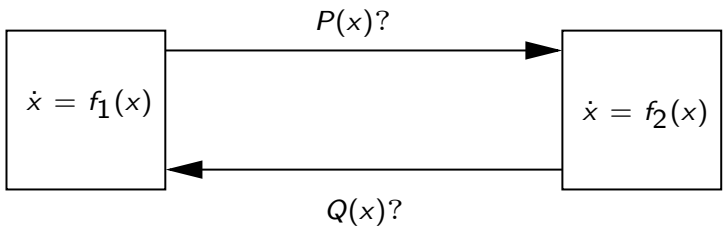
- Most systems are **embedded** in the physical environemnt via sensors and actuators
- Sometimes it is sufficient to abstract the dynamics of the environment using discrete events (the physical part of the coffee machine emits `drink-ready` sometime after receiving `st-coffee`).
- Sometime we want to estimate the **time** between the two events
- Sometime we want to look **even closer** and model how the water temperature changes over time

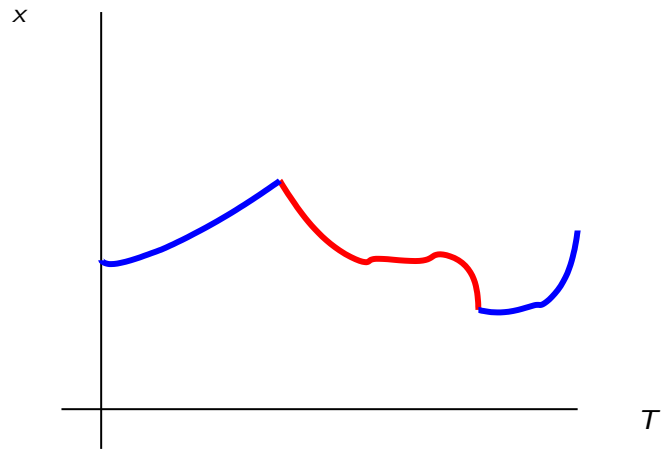
Hybrid Systems: Modeling the Physical Environment

- Most systems are **embedded** in the physical environemnt via sensors and actuators
- Sometimes it is sufficient to abstract the dynamics of the environment using discrete events (the physical part of the coffee machine emits `drink-ready` sometime after receiving `st-coffee`).
- Sometime we want to estimate the **time** between the two events
- Sometime we want to look **even closer** and model how the water temperature changes over time
- The common models for describing the dynamics of such phenomena are, alas, **continuous** and based on formalisms such as differential equations
- A new model is needed for combining **discrete** and **continuous** dynamics

Hybrid Automata

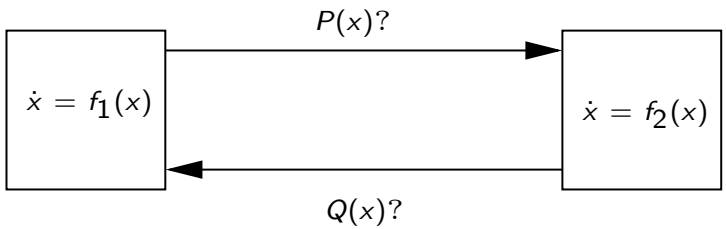
- Automata augmented with continuous variables and differential equations

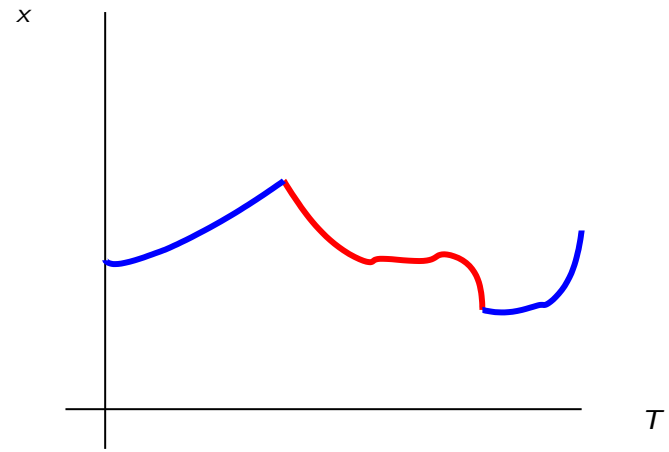




Hybrid Automata

- Automata augmented with continuous variables and differential equations





- At each state (mode) the continuous variables are subject to a specific dynamics
- When some conditions on the variables are met (or due to external event) the system may make a transition (mode switching)
- Traditional methods for analyzing continuous (linear) systems do not work for hybrid automata

Exporting Verification to Continuous (and Hybrid) Systems

- Many problems:

Exporting Verification to Continuous (and Hybrid) Systems

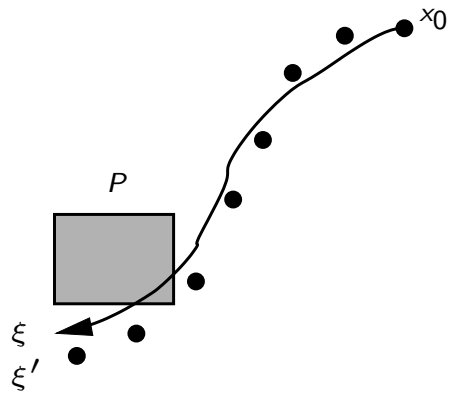
- Many problems:
- State space is $\mathbb{R}^n$, infinite even when bounded
- Time domain $\mathbb{R}$, dense and admits dirty tricks (Zeno's paradox)
- Difference between the mathematical $\mathbb{R}$ and the numerical $\mathbb{R}$ in the computer

Exporting Verification to Continuous (and Hybrid) Systems

- Many problems:
- State space is $\mathbb{R}^n$, infinite even when bounded
- Time domain $\mathbb{R}$, dense and admits dirty tricks (Zeno's paradox)
- Difference between the mathematical $\mathbb{R}$ and the numerical $\mathbb{R}$ in the computer
- How to do reachability for $\dot{x} = f(x)$ (no input)
- Sometimes we have a closed-form solution
- For $\dot{x} = Ax$, the reachable set can be written as $F_* = \{x_0 e^{At} : t \geq 0\}$
- But this description does not help us much: how to test whether $F_* \cap P = \emptyset$?

Forward Simulation for Closed Continuous Systems

- In the continuous world, numerical simulation is the dominant approach for validating systems
- Forward simulation: discretize time and replace the system with $\xi'[(n+1)\Delta] = \xi'[n\Delta] + h(\xi'[n\Delta], \Delta)$.



- This is not the "real" thing, and the simulation is not guaranteed to terminate
- But that's life and airplanes fly..

Continuous Systems with Input

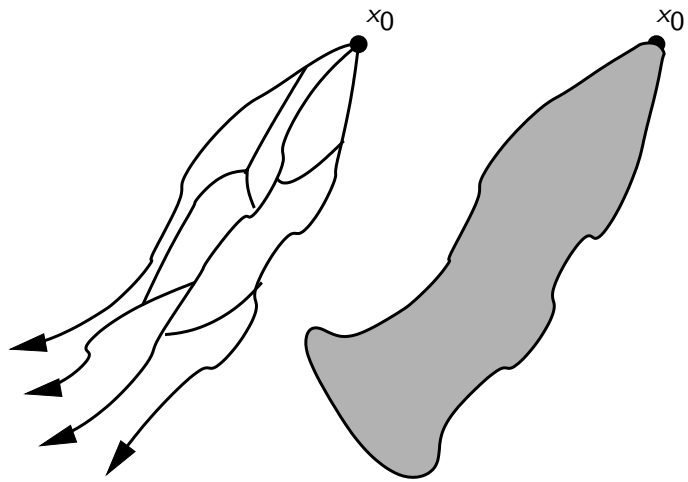
- Consider systems of the form $\dot{x} = f(x, v)$ where v is a disturbance, say, wind to the airplane
- Admissible inputs are signals of the form $\psi : T \to V$

Continuous Systems with Input

- Consider systems of the form $\dot{x} = f(x, v)$ where v is a disturbance, say, wind to the airplane
- Admissible inputs are signals of the form $\psi : T \to V$
- Problem: show that no admissible input drives the system into a set P
- For every ψ we can **simulate** and "compute" $F_*(\psi)$, but there is no finite subset of inputs that covers all reachable states

The Input Space and its Induced Behaviors

- The set of all inputs is a **doubly-dense tree**, both vertically (time) and horizontally (V).



- We are looking for ways to approximate the states reached by all these trajectories

Incremental Reachability Computation

- let $x \xrightarrow{t} x'$ denote the existence of an input signal $\psi : [0, t] \to V$ that drives the system from x to x'
- Let F be a subset of X and let I be a time interval.
- The I-successors of F are all the states that can be reached from F within that time interval, i.e.

$$\delta_I(F) = \{x' : \exists x \in F \ \exists t \in I \ x \xrightarrow{t} x'\}.$$

Incremental Reachability Computation

- let $x \xrightarrow{t} x'$ denote the existence of an input signal $\psi : [0, t] \to V$ that drives the system from x to x'
- Let F be a subset of X and let I be a time interval.
- The I-successors of F are all the states that can be reached from F within that time interval, i.e.

$$\delta_I(F) = \{x' : \exists x \in F \ \exists t \in I \ x \xrightarrow{t} x'\}.$$

- Semigroup property:

$$\delta_{[0,r_2]}(\delta_{[0,r_1]}(F)) = \delta_{[0,r_1+r_2]}(F)$$

Breadth-first Reachability Computation

$F^0 := \{x_0\}$
repeat
  $F^{k+1} := F^k \cup \delta_{[0,r]}(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$

- Two problems:
- The algorithm is not guaranteed to converge (like for most classes of infinite-state systems)
- The operator $\delta_{[0,r]}$ is not more computable than $\delta_{[0,\infty]}$ for most non-trivial systems.

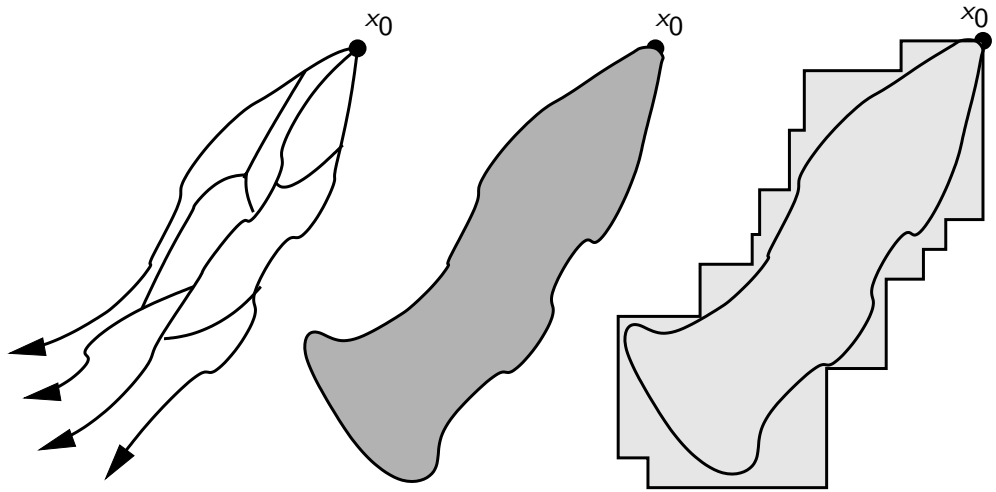
Approximate Reachability Computation

- Although $\delta_{[0,r]}(F)$ cannot be computed exactly, we can **over-approximate** it by δ' such that for every F

$$\delta_{[0,r]}(F) \subseteq \delta'_{[0,r]}(F)$$

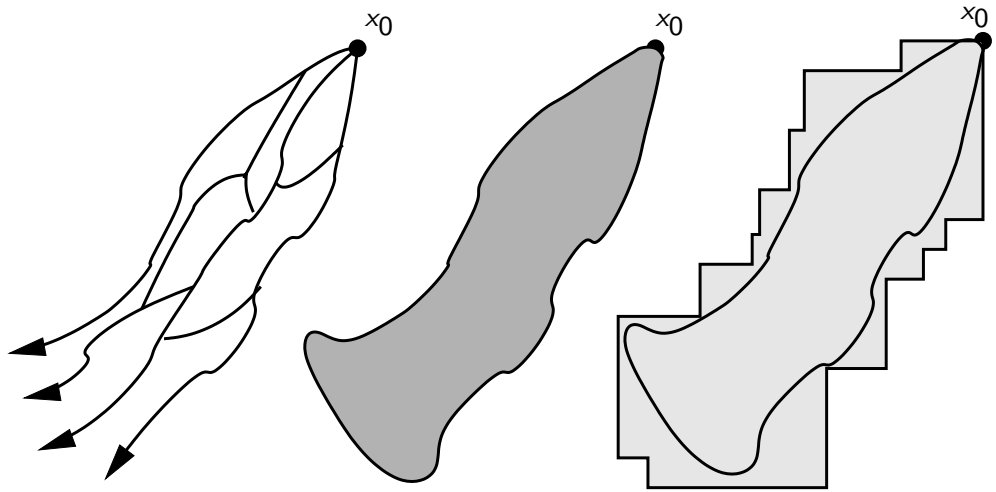
- Typicall the approximating $\delta'_{[0,r]}(F)$ belongs to some effective sub-classes of $\mathbb{R}^n$, e.g. polytopes, ellipsoids, zonotopes
- The result of the algorithm is a set F'_* s.t. $F_* \subseteq F'_*$ and hence $F'_* \cap P = \emptyset$ implies the correctness of the system

Approximate Reachability Computation - Illustration



- Numerous prototype tools were developed in the last decade to do this computation
- So far toy problems (few continuous variables, not many discrete ones)
- The only scalable techninque is of Girard and Le Guernic (2006) which can handle purely-continuous linear systems with hundreds of state variables

Approximate Reachability Computation - Illustration



- Numerous prototype tools were developed in the last decade to do this computation
- So far toy problems (few continuous variables, not many discrete ones)
- The only scalable techninque is of Girard and Le Guernic (2006) which can handle purely-continuous linear systems with hundreds of state variables
- Work is still in progress
- Other people try to focus on huge discrete state space and few real variables

Conclusions

- The “right” model to understand what’s going on in a system is a model of a dynamical system:
- State-variables, and a dynamics that describes the possible future evolutions from each state

Conclusions

- The "right" model to understand what's going on in a system is a model of a dynamical system:
- State-variables, and a dynamics that describes the possible future evolutions from each state
- Programmers and other engineers may not agree...
- Such models generate behaviors, trajectories in the state-space, that can be evaluated according to correctness or other performance measures.
- Within these models we can formulate all sorts of system design problems concerning correctness and performance

Conclusions

- The "right" model to understand what's going on in a system is a model of a dynamical system:
- State-variables, and a dynamics that describes the possible future evolutions from each state
- Programmers and other engineers may not agree...
- Such models generate behaviors, trajectories in the state-space, that can be evaluated according to correctness or other performance measures.
- Within these models we can formulate all sorts of system design problems concerning correctness and performance
- Syntax (logic assertions, programming languages) might be important for computational considerations and maybe more natural for certain engineers
- But it should not obscure the underlying dynamic semantics