automata-theoretic model checking

Kousha Etessami

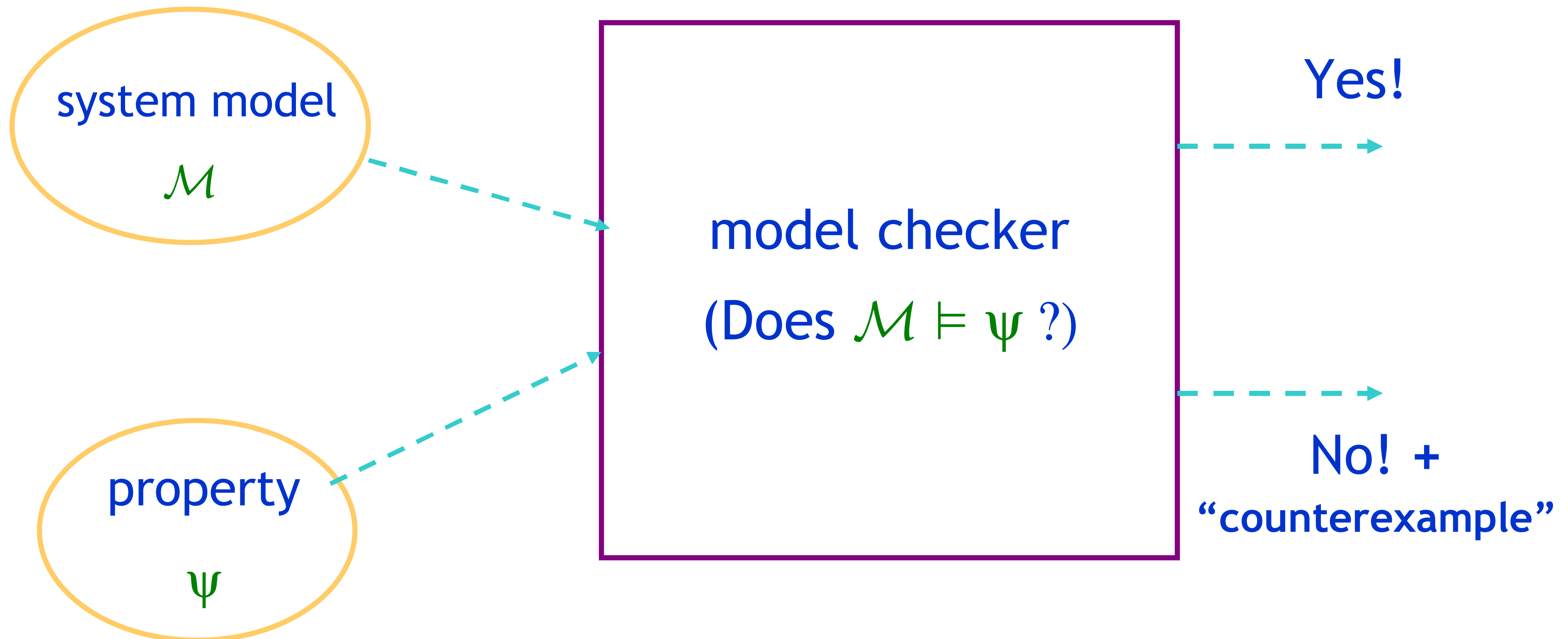
Bell Labs

(starting next week, my affiliation will be *U. of Edinburgh*)

overview

- The purpose of my lectures:
  - to cover the fundamental algorithms used in the automata-theoretic approach to (LTL) model checking, as practiced in tools such as SPIN.
- In doing so, we must cover key models and formalism: transition systems, Büchi automata, (linear) temporal logic, ……….
- We will only briefly touch on other approaches and more advanced algorithms.
- Apology: I won't provide thorough references to original research. Please see, e.g., references in the book [CGP'00].

basic picture of model checking

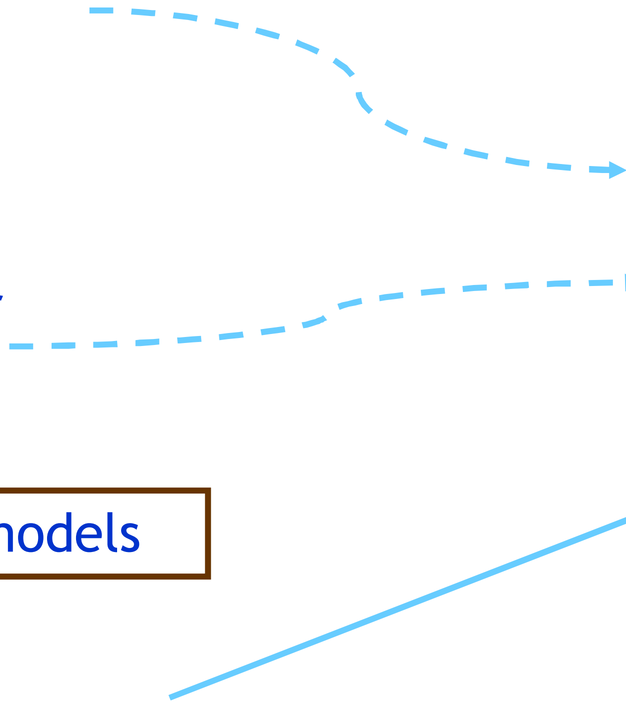


where do we get the system model?

hardware

e.g., Verilog or VHDL, source code

abstraction & other (semi-)automated transformations



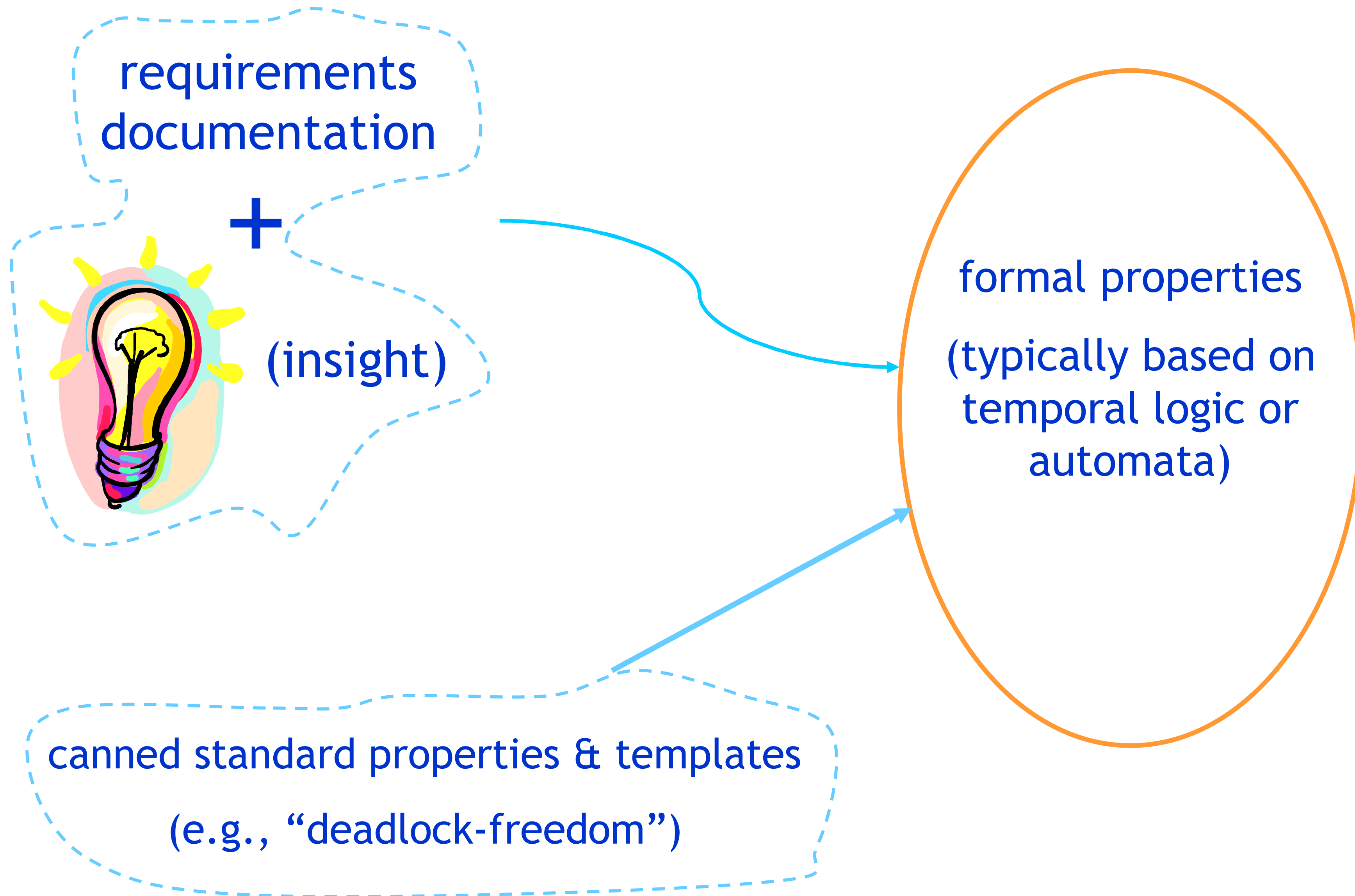
state machine-based system model

software

e.g., C, C++ , or Java, source code

hand-built design models

where do we get the properties?



choosing a system model: what do we want to model?

- systems have a **state** that evolves over time.
- they manipulate data, accessed through **variables**, whose values change as the state changes.
- **non-determinism** does not exist in real systems, but it lets us model unknowns of the system and its environment, allowing many possible evolutions of system state.
- **concurrency**: systems have interacting **processes**
  - asynchronous/synchronous,
  - message passing or shared data communication.
- possibly: dynamic memory allocation & process creation, procedure call stack, clocks and real time, etc., etc., ...
- **wait! don't get carried away....**

we need models to be:

- amenable to efficient algorithmic analysis,
- yet to faithfully model as many relevant aspects of real systems as possible.

Example: simple *Promela* implementation of a semaphore
(*Promela* is the input language of the SPIN model checker)

```
chan semaphore = [0] of { bit };
proctype dijkstra()  {
          byte count = 1;
          loop:
            if
            :: (count == 1)  → semaphore!P;   count = 0 ; goto loop
            :: (count == 0)  → semaphore?V;  count = 1 ; goto loop
            fi   }
proctype user()  {
          repeat:
               semaphore?P;  /* put critical section here …. */
               semaphore!V;   /* put non-critical section here …..*/
               goto repeat
            }
init { run dijkstra();   run user();   run user() }
```

our choice for system models: Extended Finite State Machines

An **EFSM** is given by $\mathcal{M}$ = (Q, V, Guards, Actions, Δ, init) :

- Q is a finite set of **control states**.
- $V = \{ v_1, v_2, v_3, \ldots, v_m \}$ is a set of **variables**. Each $v_i \in V$ takes values from a finite domain $D(v_i)$.
- Guards = $\{ grd_1, grd_2, \ldots, grd_r \}$, are a set of predicates $grd_i(v_1, \ldots, v_m)$, that evaluate to true or false given an assignment $\alpha \in D(v_1) \times D(v_2) \times \ldots \times D(v_m)$ to the variables.
  - guard example: "$v_2 \geq v_4$".
- Actions = $\{ act_1, act_2 , \ldots, act_d \}$ are a set of functions $act_i: D(v_1) \times \ldots \times D(v_m) \mapsto D(v_1) \times \ldots \times D(v_m)$ that given one assignment to the variables, produce another assignment.
  - action example: "$v_3 := v_1 + v_2 + 1;$".
- $\Delta \subseteq Q \times \text{Guards} \times \text{Actions} \times Q$ is a set of **transitions.**
- init = $(q_{init} , \alpha_{init})$ is an **initial** control state $q_{init} \in Q$ together with an initial assignment α_{init} to the variables.

justifying our choice

What happened to concurrency?

- No problem. If we have processes $Proc_1$, …. , $Proc_n$, with sets of control states Q_1, …., Q_n, respectively, let the product Q_1 x …. x Q_n be our set of control states Q of $\mathcal{M}$.
- in asynchronous concurrency, the transition relation Δ would modify the control state of only one process per transition.
- in synchronous concurrency, all process states could be modified in the same transition.
- bounded message buffers (“channels” in *Promela*) can be modeled by variables.

Doesn’t this hide the state explosion of the product $Q_1 \times \ldots \times Q_n$?

- Not really. Variables already yield state explosion. In fact, we can eliminate control states entirely, encoding them as variables: let variable pc_i encode the “program counter” value for process $Proc_i$. Transitions of $\mathcal{M}$ are then just pairs in Guards x Actions, that update program counters appropriately.

What about dynamic process creation and dynamic memory allocation?

- In principle, EFSMs are finite state and do not model systems whose state space can be unbounded.

But tools like SPIN allow dynamic process creation, don't they?

- Yes, but no problem. The algorithms we describe for EFSMs apply to a more general model, where the state space could be infinite, but without any guarantee of halting.
- To keep things simple, we confine our descriptions to EFSMs. This already conveys the essence of all the algorithms.
- But we will soon describe more general assumptions, sufficient for the algorithms to be applicable.

What about procedure calls? What about clocks and real time?

- See J. Esparza's and K. Larsen's lectures, respectively.

The underlying transition system of an EFSM

- An EFSM, $\mathcal{M}$, provides a concise representation of an underlying **state transition system** $\mathcal{K}$ = (S,R,init) :
  - S = Q x (D(v_1) x…x D(v_m)) is the set of (full) **states** of $\mathcal{K}$, given by control state + data values in $\mathcal{M}$.
  - $R \subseteq S \times S$, the transitions of $\mathcal{K}$, are:

    for all states $s = (q,\alpha) \in S$ and $s' = (q', \alpha') \in S$

    $(s,s') \in R$ iff $\exists$ (q, grd , act, q′) $\in \Delta$ such that

    grd(α) = true and act(α) = α' .
  - init = $(q_{init}, \alpha_{init}) \in S$, is also the initial state of $\mathcal{K}$.
- Each execution of $\mathcal{M}$ defines an (infinite) path π, called a **run** through states in $\mathcal{K}$, where $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots\ldots$ such that s_0 = init, and $(s_i,s_{i+1}) \in R$, $\forall\, i \geq 0$.

labeling the transition system

- Suppose we're interested in certain "observations" or "propositions" $P = \{ p_1, \ldots, p_k \}$ regarding states of $\mathcal{K}$. Each

  $$p_i : S \mapsto \{\text{true}, \text{false}\}$$

  is a predicate on states

  (i.e., a predicate on control state + data values in $\mathcal{M}$).

  - Examples: "counter $\geq$ 1"

    "the control-state is q or else c = 0"

- We can label state s with the propositions $L(s) \subseteq P$ that hold true at s. Thus, $L: S \mapsto \Sigma_P$, where the alphabet $\Sigma_P = 2$^P is the set of subsets of P.

- We thus get a labeled transition system (**Kripke structure**)

  $$\mathcal{K}_P = \quad (S\ ,\ \Sigma_P\ ,\ R\ ,\ L\ ,\ \text{Init})$$

associating an ω-language with an EFSM

- For a run $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \ldots\ldots$ of $\mathcal{K}_P$, we let

$$L(\pi) \triangleq L(s_0)\, L(s_1) \ldots. \in (\Sigma_P)^\omega$$

  be the ω-**word** associated with the run.

- We define $L_P(\mathcal{M}) \triangleq L(\mathcal{K}_P) \triangleq \{\, L(\pi) \mid \pi \text{ is a run of } \mathcal{K}_P \}$ to be the ω-**language** associated with $\mathcal{M}$ (with respect to proposition set P) .

assumptions about $\mathcal{K}$ for our algorithms & their analysis

- We assume S is at most countable, and hence each $s \in S$ has a finite description. (For EFSMs, S is finite.)
- For each $s \in S$,
  - we assume there are a finite set of successor states $Succ(s) = \{s' \mid (s,s') \in R\}$, and that you can "efficiently" compute them.
  - Thus, Guards & Actions are assumed to be "efficiently" computable.
  - We also assume each proposition $p_i \in P$ (a predicate on states s) is "efficiently" computable.

A quick detour: Büchi automata

- If we add a set $F \subseteq S$ of “accepting” states to a Kripke structure we get a **Büchi automaton**:

$$\mathcal{A} = (S , \Sigma , R , L , Init, F)$$

- A run $\pi = s_0 \to s_1 \to s_2 \to \ldots\ldots$ of $\mathcal{A}$ is now called an **accepting** run iff for infinitely many $i \in \mathbb{N}$, $s_i \in F$.
- We’ll allow a set of initial states $Init \subseteq S$, rather than just one. A run π must now have $s_0 \in Init$.
- We associate with $\mathcal{A}$ the ω-language

$$L(\mathcal{A}) = \{ L(\pi) \mid \pi \text{ is an accepting run of } \mathcal{A}\}.$$

- We say ω-word $w \in \Sigma^\omega$ is accepted by $\mathcal{A}$ if $w \in L(\mathcal{A})$.

- In a **generalized** Büchi automaton, instead of one accepting set F, we have a family $\mathcal{F} = \{ F_1, \ldots, F_d \}$, and π is accepting iff for all $F \in \mathcal{F}$, for infinitely many $i \in \mathbb{N}$, $s_i \in F$.

specification of properties: temporal logic

- What is temporal logic?
  - It is a language for describing relationships between the occurrence of events over time.
  - It comes is several flavors, e.g., Linear vs. Branching time. We focus on (propositional) **Linear Temporal Logic** (**LTL**) .

  - **Example 1**:

    "Until event stop occurs, every occurrence of event request is eventually followed by an occurrence of event response":

    (request $\rightarrow$ "eventually" response) "Until" stop

    In LTL syntax:

    $$(\text{request} \rightarrow \Diamond\, \text{response})\ \mathrm{U}\ \text{stop}$$

  - **Example 2**:

    "infinitely often pulse": "always" "eventually" pulse

    In LTL syntax: $\Box \Diamond$ pulse

Why use temporal logic to specify properties?

- [Pnueli'77] and others recognized that correctness assertions for **reactive** systems are best phrased in terms of occurrence of events during the entire, potentially indefinite, execution of the system. Not just what it outputs when it halts.

- Indeed, systems like the *Windows OS* aren't really supposed to "halt and produce output". Rather, they should forever **react** to stimuli from their environment in a "*correct*" manner.

- There exist properties that we might wish to express that aren't expressible in LTL, nor in other temporal logics.
- But we need to balance expressiveness of the specification language with algorithmic efficiency of checking such properties.
- temporal logics, and the related formalism of ω-automata, provide a reasonable balance.
- more on LTL vs. branching-time logics later.....

syntax of LTL

- LTL formulas (with base propositions P) are built from:
  - atomic propositions : $P = \{ p_1, \ldots, p_k \}$,
  - boolean connectives: $\{ \neg , \vee , \wedge \}$,
  - temporal operators: $\{ \bigcirc , \Diamond , \Box , \mathrm{U} , \mathrm{V} \}$.

  (we also use parentheses to disambiguate formulas)
- Inductively, the set of LTL formulas are:
  - p_i, for all $p_i \in P$.
  - if ψ_1 and ψ_2 are LTL formulas, then so are

  $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\bigcirc \psi_1$, and $\psi_1 \, \mathrm{U} \, \psi_2$.
- The other connectives & operators are expressively redundant:

$$\psi_1 \, \mathrm{V} \, \psi_2 \equiv \neg(\neg \psi_1 \, \mathrm{U} \, \neg \psi_2) \, ,$$

$$\Diamond\psi \equiv \text{true} \, \mathrm{U} \, \psi, \qquad \text{(where true} \triangleq p_1 \vee \neg p_1 \text{)}$$

$$\Box \psi \equiv \neg \Diamond \neg \psi$$

semantics of LTL

- We interpret a formula ψ as expressing a property of ω-words, i.e., an ω-language $L(\psi) \subseteq (\Sigma_P)^\omega$.
- For ω-word $w = w_0 w_1 w_2 \ldots\ldots \in (\Sigma_P)^\omega$, let $w[i] = w_i w_{i+1} w_{i+2} \ldots\ldots$ be the suffix of w starting at position i. We define the "satisfies" relation, $\models$, inductively:
  - $w \models p_j$ iff $p_j \in w_0$ (for any $p_j \in P$).
  - $w \models \neg\psi$ iff $w \not\models \psi$.
  - $w \models \psi_1 \vee \psi_2$ iff $w \models \psi_1$ or $w \models \psi_2$.
  - $w \models \bigcirc\psi$ iff $w[1] \models \psi$.
  - $w \models \psi_1 \, U \, \psi_2$ iff $\exists\, i \geq 0$ such that $w[i] \models \psi_2$, & $\forall j,\ 0 \leq j < i,\ w[j] \models \psi_1$.
- Let $L(\psi) = \{ w \mid w \models \psi \}$.

semantics of LTL continued…………

Extending our definition of "satisfies" to transition systems, and EFSMs, we say:

- $\mathcal{K} \models \psi$ iff for all runs π of $\mathcal{K}_P$, $L(\pi) \models \psi$,
  i.e., $L(\mathcal{K}_P) \subseteq L(\psi)$.

- $\mathcal{M} \models \psi$ iff $\mathcal{K} \models \psi$ (where $\mathcal{K}$ is the underlying transition system of $\mathcal{M}$.)

a few words about branching-time TL's

- Branching-time (BT) temporal logics allow us to speak about many executions of $\mathcal{M}$ at the same time, using quantification over paths π.
- Example: *"In all executions, at all times there is a a way to reach the 'reset' state"*, would in the branching-time logic CTL be expressed as:

$$A \Box E \Diamond \text{reset}$$

- Thus, unlike LTL, we interpret BT logics not on ω-words, but on ω-**trees**, consisting of all runs rooted at the initial state, labeled by Σ_P.

linear vs. branching-time logics

some advantages of LTL

- LTL properties are preserved under "abstraction": i.e., if $\mathcal{M}$ "approximates" a more complex model $\mathcal{M}'$, by introducing more paths, then
  $$\mathcal{M} \models \psi \Rightarrow \mathcal{M}' \models \psi$$
- "counterexamples" for LTL are simpler: consisting of single executions (rather than trees).
- The automata-theoretic approach to LTL model checking is simpler (no tree automata involved).
- anecdotally, it seems most properties people are interested in are linear-time properties.

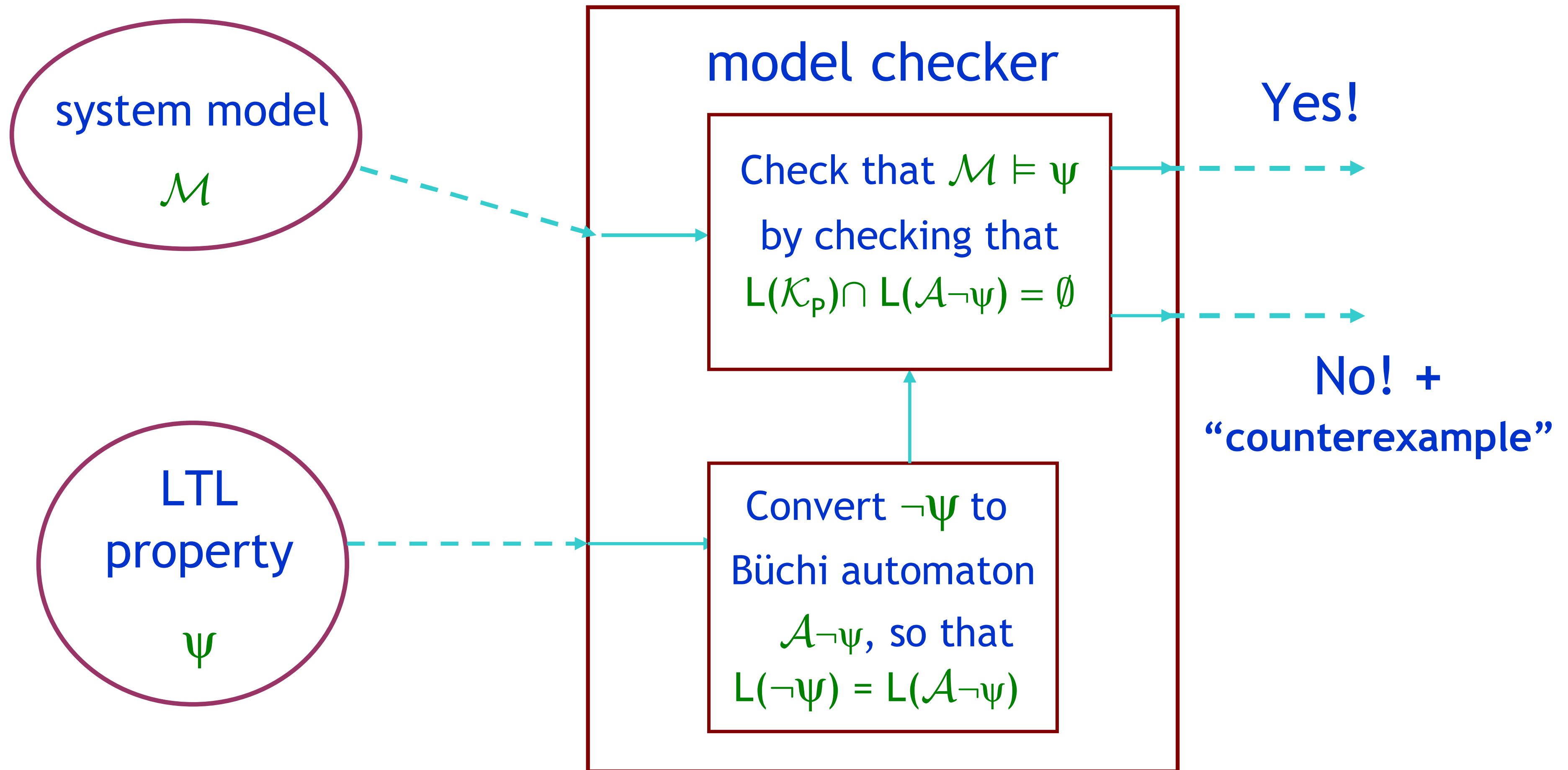
some advantages of BT logics

- BT allows expression of some useful properties like 'reset'.
- CTL, a limited fragment of the more complete BT logic CTL*, can be model checked in time linear in the formula size (as well as in the transition system). But formulas are usually far smaller than system models, so this isn't as important as it may first seem.
- Some BT logics, like μ-calculus and CTL, are well-suited for the kind of fixed-point computation scheme used in symbolic model checking.

My bias: linear: LTL and more generally ω-regular properties on words.

The automata-theoretic approach to LTL model checking

[Vardi-Wolper'86]:
(informed by much prior work: [C-E'81,Q-S'81,L-P'85,K'85,.....])



why does this scheme work?

$$\begin{aligned}
\mathcal{M} \models \psi \quad &\Leftrightarrow \quad L_P(\mathcal{M}) \subseteq L(\psi) \\
&\Leftrightarrow \quad L(\mathcal{K}_P) \subseteq L(\psi) \\
&\Leftrightarrow \quad L(\mathcal{K}_P) \cap (\Sigma_P{}^{\omega} \setminus L(\psi)) = \emptyset \\
&\Leftrightarrow \quad L(\mathcal{K}_P) \cap L(\neg\psi) = \emptyset \\
&\Leftrightarrow \quad L(\mathcal{K}_P) \cap L(\mathcal{A}_{\neg\psi}) = \emptyset \,.
\end{aligned}$$

our algorithmic tasks

We have reduced LTL model checking to two tasks:

1. Convert an LTL formula φ (=$\neg\psi$) to a Büchi automaton $\mathcal{A}_\varphi$, such that $L(\varphi) = L(\mathcal{A}_\varphi)$.

    - Can we in general do this? yes……

2. Check that $\mathcal{M} \vDash \psi$, by checking that the intersection of languages $L(\mathcal{K}_P) \cap L(\mathcal{A}_{\neg\psi})$ is empty.

    - It would be unwise to first construct all of $\mathcal{K}_P$ from $\mathcal{M}$, because $\mathcal{K}$ can be far too big (*state explosion*).
    - Instead, we shall see how to construct states of $\mathcal{K}_P$ only as needed.

Translating LTL to Büchi automata: basic ideas

- First, let's put LTL formulas φ in **<u>normal form</u>** where:
  - $\neg$'s have been "pushed in", applying only to propositions.
  - the only temporal operators are U and its dual, V.
- States of $\mathcal{A}_\varphi$ will be <u>sets of subformulas</u> of φ, thus if we have $\varphi = p_1 \; U \; \neg p_2$, a state is given by $\Phi \subseteq \{ p_1 , \neg p_2 , p_1 U \neg p_2 \}$.
- Consider a word $w = w_0 \, w_1 \, w_2 \, ...$, such that $w \models \varphi$, where, e.g., $\varphi = \psi_1 U \psi_2$.
- Mark each position i with the set of subformulas Φ_i of φ that hold true there:

$$\Phi_0 \; \Phi_1 \; \Phi_2 \; ....$$
$$w_0 \; w_1 \; w_2 \; ....$$

- Clearly, $\varphi \in \Phi_0$. But then, by <u>consistency</u>, either:
  - $\psi_1 \in \Phi_0$, and $\varphi \in \Phi_1$, or
  - $\psi_2 \in \Phi_0$.
- Such consistency rules will dictate our states & transitions ....

The translation: LTL to (generalized) Büchi

Let sub(φ) denote the set of subformulas of φ.

We define $\mathcal{A}\varphi$ = (Q, Σ , R, L, Init, $\mathcal{F}$) as follows.

First, the state set:

- Q = { $\Phi \subseteq$ sub(φ) | s.t. Φ is internally consistent }.
  - For Φ to be internally consistent we should, e.g., have:
    - if $\psi \vee \gamma \in \Phi$, then $\psi \in \Phi$ or $\gamma \in \Phi$.
    - if $\psi \wedge \gamma \in \Phi$, then $\psi \in \Phi$ and $\gamma \in \Phi$.
    - if $p \in \Phi$ then $\neg p \notin \Phi$, & if $\neg p \in \Phi$ then $p \notin \Phi$.
    - if ψ U $\gamma \in \Phi$, then ($\psi \in \Phi$ or $\gamma \in \Phi$).
    - if ψ V $\gamma \in \Phi$, then $\gamma \in \Phi$.
- (some of these are not strictly necessary, but we get a smaller automaton by using them to eliminate redundant states).

the translation continued…….

Now, labeling the states of $\mathcal{A}\varphi$:

- The alphabet is $\Sigma = 2^{P \cup \neg P}$, where $\neg P = \{ \neg p \mid p \in P \}$.
- The labeling, $L: Q \mapsto \Sigma$, is $L(\Phi) = \{ l \in P \cup \neg P \mid l \in \Phi \}$.
- <u>**Note**</u>: $\sigma = \{ l_1,\ldots,l_m \} \in \Sigma$ denotes a <u>boolean term</u> (a conjunct of literals) $l_1 \wedge \ldots \wedge l_m$, <u>not</u> one symbol of $\Sigma_P = 2^P$.
  - Now, a word $w = w_0 w_1 \ldots \in (\Sigma_P)^\omega$ is in $L(\mathcal{A}\varphi)$ iff there's a run $\pi = \Phi_0 \rightarrow \Phi_1 \rightarrow \Phi_2 \ldots$ of $\mathcal{A}\varphi$, s.t., $\forall i \in \mathbb{N}$, w_i "<u>satisfies</u>" $L(\Phi_i)$, i.e., is a "satisfying assignment" for the term $L(\Phi_i)$.
  - This constitutes a slight redefinition of Büchi automata, but it is important for facilitating a much more compact $\mathcal{A}\varphi$.

translation continued....

Now, the transition relation, and the rest of $\mathcal{A}\varphi$:

- $R \subseteq Q \times Q$, where $(\Phi, \Phi') \in R$ iff:
  - if $\bigcirc\psi \in \Phi$, then $\psi \in \Phi'$.
  - if $\psi U \gamma \in \Phi$ then $\gamma \in \Phi$, or ($\psi \in \Phi$ and $\psi U \gamma \in \Phi'$).
  - if $\psi V \gamma \in \Phi$ then $\gamma \in \Phi$, and $\psi \in \Phi$ or $\psi V \gamma \in \Phi'$.
- Init = $\{ \Phi \in Q \mid \varphi \in \Phi \}$.
- For each $\chi \in sub(\varphi)$ of the form $\chi = \psi U \gamma$, there is a set $F_\chi \in \mathcal{F}$, such that $F_\chi = \{ \Phi \in Q \mid \chi \notin \Phi$ or $\gamma \in \Phi\}$.

Lemma: $L(\varphi) = L(\mathcal{A}\varphi)$.

but, at this point $\mathcal{A}\varphi$ is a generalized Büchi automaton.....

From generalized-Büchi automata to Büchi automata

From $\mathcal{A}$ = (Q, Σ , R, L, Init, $\mathcal{F}$ = {F_0, …, F_d}) ,
we construct $\mathcal{A}'$ = (Q', Σ , R', L', Init', F') :

- Q' = Q x {0, …,d} .
- R' $\subseteq$ Q' x Q', where ((q,i),(s,j)) $\in$ R' iff (q,s) $\in$ R, & either q $\notin$ F_i and i=j, or q$\in F_i$ and j=(i+1) mod (d+1) .
- L'((q,i)) = L(q).
- Init' = { (q,0) | q $\in$ Init }.
- F' = { (q,0) | q $\in$ F_0 }.

<u>**Lemma**</u>: L($\mathcal{A}'$) = L($\mathcal{A}$) .

We have shown:

THEOREM: *Every LTL formula* φ *can be converted to a Büchi automaton* $\mathcal{A}_\varphi$*, such that* $L(\varphi)=L(\mathcal{A}_\varphi)$. *Furthermore,* $|\mathcal{A}_\varphi| \in 2^{O(|\varphi|)}$.

We need $\mathcal{A}_\varphi$ to be as small as possible. But it is PSPACE-hard to find an optimal $\mathcal{A}_\varphi$.

- [GPVW'95] give a more efficient algorithm, constructing only states as needed, in practice yielding smaller automata. Related translations based on alternating-automata,[MSS'88,V'94,GO'01], yield further improvements.
- Beyond optimizing the translation itself, key optimizations are:
  - [EH'00, SB'00]:"massage" LTL formulas before translation, using "proof-theoretic" rewrite rules.
  - [EH'00,SB'00, EWS'01,E'02]: make Büchi automata smaller after translation, using various "fair" simulation quotients......
- In the worst case, exponential blowup is unavoidable. Consider:

$$\Diamond p_1 \wedge \Diamond p_2 \wedge \ldots\ldots \wedge \Diamond p_n$$

Can we go in the other direction? i.e., can every Büchi automaton $\mathcal{A}$ be converted to an LTL formula $\psi_{\mathcal{A}}$, such that $L(\mathcal{A}) = L(\psi_{\mathcal{A}})$?

- No, not quite. But there are relatively simple modifications to LTL ([Wolper'83]) that make it as expressive as Büchi automata.
- For example, LTL formulas prefixed by quantification of one proposition, $\exists p\, \psi$, suffice to express all ω-regular languages ([T'82,E'00]).

What about ω-automata with other acceptance criteria? Rabin, Streett, Muller, ……………..

what's with all these names?

- Each such acceptance condition simply defines a different family of <u>boolean formulas</u> over predicates inf(q) meaning "*state q occurs infinitely often in the run*".

Person's name	Class of acceptance formulas
Büchi	$\bigvee_{q \in F} \; inf(q)$
(generalized) Büchi	$\bigwedge_{F \in \mathcal{F}} \bigvee_{q \in F} \; inf(q)$
Rabin	$\bigvee_{i \in [k]} (\neg (\bigvee_{q \in Li} \; inf(q)) \; \wedge \; (\bigvee_{i \in Ri} \; inf(q)))$
Streett	$\bigwedge_{i \in [k]} (\quad (\bigvee_{q \in Li} \; inf(q)) \; \rightarrow \; (\bigvee_{i \in Ri} \; inf(q)))$
Muller	truth table shorthand for arbitrary boolean formulas
your name here	your favorite formulas (with desirable properties)

- **<u>Theorem:</u>** all of these can be converted to equivalent (non-deterministic) Büchi automata. (but translation costs vary.)

step 2: determining whether $L(\mathcal{K}_P) \cap L(\mathcal{A}\varphi) = \emptyset$:
first, the basic product construction

Given $\mathcal{K}_P$ = (S,R,L, init) and $\mathcal{A}\varphi$ = (Q,Σ_P,R',L',Init',F'),
we define the "product" Büchi automaton:

$$\mathcal{K}_P \otimes \mathcal{A}\varphi = (S^\otimes, \Sigma_P, R^\otimes, L^\otimes, Init^\otimes, F^\otimes)$$

$S^\otimes$:= { $(s,q) \in S \times Q$ | L(s) satisfies L'(q) } ,

(recall: L'(q) is a *term*)

$R^\otimes$:= { $((s,q),(s',q')) \in S^\otimes \times S^\otimes$ | $(s,s') \in R$ & $(q,q') \in R'$ },

$L^\otimes((s,q))$:= L(s),

$Init^\otimes$:= { $(init,q) \in S^\otimes$ | $q \in Init'$ } ,

$F^\otimes$:= { $(s,q) \in S^\otimes$ | $q \in F'$ } .

Fact 1: $L(\mathcal{K}_P \otimes \mathcal{A}\varphi) = L(\mathcal{K}_P) \cap L(\mathcal{A}\varphi)$.

But how do we determine whether $L(\mathcal{K}_P \otimes \mathcal{A}\varphi) = \emptyset$?

determining whether $L(\mathcal{K}_P) \cap L(\mathcal{A}\varphi) = \emptyset$, continued......

Fact 2: $L(\mathcal{K}_P \otimes \mathcal{A}\varphi) = \emptyset$ iff there is no reachable cycle in the Büchi automaton $\mathcal{K}_P \otimes \mathcal{A}\varphi$ containing a state in $F^{\otimes}$.

So, it appears we are left only with the task of finding whether there is such a reachable cycle.

But, NOTE: we are not given $\mathcal{K}_P$. We're given $\mathcal{M}$!
We won't first build $\mathcal{K}_P$ from $\mathcal{M}$, because $\mathcal{K}_P$ can be HUGE!!

on-the-fly bad cycle detection

- Given $\mathcal{M}$ and $\mathcal{A}\varphi$, we will explore the state space of $\mathcal{K}_P \otimes \mathcal{A}\varphi$, but only constructing states as we need them. If we find a bad cycle, we stop before exploring all of $\mathcal{K}_P \otimes \mathcal{A}\varphi$.
- We begin with the set of initial states $Init^{\otimes}$, which we can obtain directly from $\mathcal{M}$ and $\mathcal{A}\varphi$.
- For $s \in S^{\otimes}$, by assumption we can compute $Succ^{\otimes}(s)$ efficiently, so we can do Depth First Search, to find all reachable states of $\mathcal{K}_P \otimes \mathcal{A}\varphi$.
- But how do we find bad cycles efficiently?
  - We could simply compute the SCCs of $\mathcal{K}_P \otimes \mathcal{A}\varphi$ using the standard DFS algorithm, and check if $\exists$ a reachable (nontrivial) SCC containing a state of $F^{\otimes}$.
  - But this is too inefficient in practice. We will use a cool nested DFS [CVWY'90].

The [CourcobetisVardiWolperYannakakis'90] algorithm

```
Input: 𝓜 and 𝓐φ,
Initialize:  Stack1:=∅ , Stack2:=∅ ,
             Table1:=∅ , Table2:=∅ ;
procedure Main() {
  foreach  s ∈ Init⊗
   { if  s ∉ Table1  then DFS1(s); }
  output("no bad cycle");
  exit;
}
procedure DFS1(s)  {
   push(s,Stack1);
   hash(s,Table1);
   foreach  t ∈ Succ⊗(s)
    { if t ∉ Table1 then DFS1(t); }
   if  s ∈ F⊗ then {  DFS2(s);  }
   pop(Stack1);
}
```

```
procedure DFS2(s){
  push(s,Stack2);
  hash(s,Table2) ;
  foreach t ∈ Succ⊗(s) do {
    if t ∉ Table2 then
            DFS2(t)
    else  if t is on Stack1 {
     output("bad cycle:");
     output(Stack1,Stack2,t);
     exit;
     }
   }
   pop(Stack2);
}
  /*note: upon finding a bad cycle,
     Stack1,Stack2,+t, determines
     a counterexample: a bad cycle
     reached from an init state.*/
```

Theorem: The [CVWY'90] algorithm outputs "no bad cycle" iff $L(\mathcal{K}_P) \cap L(\mathcal{A}\varphi) = \emptyset$. If it outputs "bad cycle:…", then content of Stack1 + Stack2 + t defines a (looping) execution of $\mathcal{M}$ that does not satisfy property φ.
The algorithm (properly implemented) runs in (worst-case) time and space: $O(|\mathcal{K}_{\mathcal{P}}| \times |\mathcal{A}\varphi|)$, i.e. , $O(2^{O(|\mathcal{M}|)} \times |\mathcal{A}\varphi|)$.

- If we find a bad cycle, we can get lucky and find it much faster than the worst-case time.
- The crux of the *state explosion problem* is the $2^{O(|\mathcal{M}|)}$ factor.
- Note that we are better off having $\mathcal{A}\varphi$ as small as possible before running this check.

methods for mitigating the state explosion problem

We'll look briefly at:

- Bit-state hashing.
- Partial-order reduction.

We will not look at:

- Symbolic model checking: See A. Cimatti's lecture.
  idea: view R compactly as a boolean formula $R(b_1,\ldots,b_n,b'_1,\ldots,b'_n)$, represented & manipulated as an **OBDD**.
- Abstraction & Compositional model checking:
  See M. Huth's & L. de Alfaro's lectures, respectively.
- Symmetry considerations:
  idea: often, a system consists of related or identical components. Exploit non-trivial symmetries in state space to restrict the search space. (See, e.g., the book [CGP'00].)
- SAT-based bounded model checking: (see [BCCZ'99],…..)
  idea: construct a boolean formula that is satisfiable iff "there is a counterexample to $\mathcal{M} \models \psi$ of 'length' at most k", and check its satisfiability using a SAT-solver.

bit-state hashing [H’88]

- Ordinarily, collisions in hash tables (used, e.g., in the [CVWY’90] algorithm) are resolved by, e.g., chaining or open addressing.
- In practice, this has a high cost.
- “Bit-state hashing” reduces this cost by having each location of the hash table contain only one bit. If any state hashes to that location, the bit is set to 1.
- Thus, because of collisions, the algorithm might over-approximate what states it thinks have already been processed.
- Hence, some of the state space might not be searched, and errors (bad cycles) might be missed.
- But experiments suggest that variations on this scheme ([H’88,GH’93]) perform reasonably well in many circumstances.

partial-order reduction

[V'90],[GW'91],[P'93],[HP'94],……

- In an asynchronous concurrent system
  $\mathcal{M} = Proc_1 \| \ldots. \| Proc_n$, with control states $Q_1 \times \ldots. \times Q_n$, sometimes actions occuring on different processes are independent, and can be commuted without changing any relevant fact about the execution.

  Example: if $Proc_1$ and $Proc_2$ each increment a local counter, we might not care in what order this happens.

- The idea of partial-order reduction is to group together executions that differ merely by commutations of independent transitions, and try to confine the search to only explore one representative from each such "equivalence class".
- There are several techniques for choosing a subset of transitions during, e.g., nested DFS search, so that these transitions will "cover" all equivalence classes of executions, with hopefully little redundancy.
- There are many details involved. See the above references and [CGP'00].

review

We have:

- defined EFSMs as system models, and defined their underlying transition systems.
- defined (linear) temporal logic (LTL) as a specification formalism.
- defined Büchi automata.
- provided an algorithm for model checking an EFSM against an LTL specification, using:
  - an algorithm to translate an LTL formula to a Büchi automaton.
  - an “on-the-fly” algorithm to determine if the intersection of the language associated with an EFSM and that of a Büchi automaton is empty.
- discussed some approaches to coping with state explosion.

The END

(I rue the day I decided to use PowerPoint for this.)