Q: What kind of systems can we verify?

A: systems whose behavior is described precisely \(\Rightarrow\) mathematically
states and transitions (informally: “circles and arrows”)

One of the simplest models: finite state machine

Another view: system state: set of all quantities that determine the behavior of the system in time

Representation: every state has unique binary encoding (state variable)

Definition of state: depends on abstraction level
Example for a processor: instruction set level; internal organization (incl. pipeline); register transfer level; gate-level; transistor level
- discrete, continuous or hybrid systems
- finite (\(\Rightarrow\) must be discrete) or infinite (continuous systems; programs with recursion or dynamic data structures)

Finite state machines (automata): defined by states and transitions
- ex. program state = variables + prog. counter; transitions = statements
- (finite state if finite types, no recursion, no dynamic data)

Our model: a set \(V = \{v_1, v_2, \ldots, v_n\}\) of variables over a domain \(D\)
- a state: an assignment \(s: V \rightarrow D\) of values for each variable in \(D\)
- A state (assignment) \(\Leftrightarrow\) a formula true only for that assignment
  \((v_1 = 7) \land (v_2 = 4) \land (v_3 = 2)\)
- A formula \(\Leftrightarrow\) the set of all assignments that make it true
  \(\Rightarrow\) sets of states: representable by logic formulas, e.g., \(v_1 \leq 5 \land v_2 > 3\)
- A transition \(s \rightarrow s'\) has two states \(\Rightarrow\) a formula over \(V \cup V'\)
  where \(V' = \text{copy of } V\) (next state variables)
  e.g.,\((\text{semaphore = red}) \land (\text{semaphore' = green})\)
- Transition relation: set of all transitions = a formula \(R(V, V')\)

Transitions are given as a relation, not a function.
\(\Rightarrow\) there can be several states \(s'\) such that \(s \rightarrow s', \text{ i.e., } (s, s') \in R\)
In this case the model (Kripke structure) is called nondeterministic
(the future behavior in a state is not uniquely determined).

This is different from the DFA / NFA distinction: finite state automata have transitions labeled with input symbols
\(\Rightarrow\) deterministic if unique next state for given state and input symbol (even if different inputs can lead to different states)

For systems viewed as open (interacting with an environment), this is called input nondeterminism
Typically, we view Kripke models as closed; we will discuss possible parallel composition with an environment
**Linear Temporal Logic (LTL)**

Defined by Amir Pnueli in 1977 (ACM Turing Award 1996). Describes event sequencing along an execution path ⇒ *linear* structure
– an event happens in the future
– a property is invariant (holds everywhere) starting at a given state
– an event follows another event

*Temporal operators* (truth modalities along an execution trace)
- **X** (next): in the next state also written \(\Box\)
- **F** (future): sometime in the future
- **G** (globally): in every future state (including now)
  - unary operators, refer to one property
- **U** (until) binary operator, \(\text{property}_1\) until \(\text{property}_2\)

Sometimes also: release operator **R** (dual to until). Ignored here.

---

**LTL Semantics**

LTL formulas of the form \(\text{A} f\) have their meaning defined in a state ⇒ called *state formulas*: true if all paths from \(s\) satisfy \(f\)

Path formulas have their meaning (truth value) defined over a path.

Notations:
- \(\text{M}, s \models f\) in the model (Kripke structure) \(\text{M}\), state \(s\) satisfies \(f\)
- \(\text{M}, \pi \models f\) in model \(\text{M}\), path \(\pi\) satisfies \(f\)

If \(\text{M}\) is fixed (given), we simply write \(s \models f\), \(\pi \models f\)

\(s^1 =\) suffix of path \(\pi = s_0 s_1 s_2 \ldots\) starting at \(s_1 = s_0 s_1 + 1 s_2 + 2 \ldots\)

Semantics of state formulas:
- \(s \models p\) ⇔ \(p \in L(s)\) (state \(s\) has \(p\) as a label)
- \(s \models \text{A} f\) ⇔ \(\pi \models f\) for all paths \(\pi\) from \(s\)

For path formulas, define semantics as usual by *structural induction*:
the semantics of a formula is given in terms of its simpler subformulas

---

**The temporal logic CTL***

LTL is a *linear* logic: paths are viewed independently; there may be many futures from origin, but can’t express branching *at each step* ⇒ not expressive enough (e.g., always possible to reach a state) ⇒ another model: *computation trees* (branching view)

Finite unfolding of a state-transition graph starting from an initial state

---

**LTL Syntax**

Express that a property is true *for all* paths ⇒ using the *universal quantifier* \(\text{A}\)
⇒ LTL formulas are of of the form \(\text{A} f\), where \(f\) is a *path formula*

*Syntax* of path formulas:
- \(f :::= p\) base case: \(p \in \text{AP}\) an atomic proposition
- \(\neg f\) \(\lor f\) \(\land f\) usual boolean connectors
- \(\text{X} f\) \(\text{F} f\) \(\text{G} f\) \(\text{U} f\) temporal operators

Since the \(\text{A}\) quantifier is mandatory, and appears only once, it is sometimes left implicit (some authors write path formulas only)

---

**LTL semantics: path formulas**

Semantics of path formulas:
- \(\pi \models p\) ⇔ \(s \models p\) \(p \in \text{AP}\) holds in path origin
- \(\pi \models \neg f\) ⇔ \(\pi \models f\) \(\pi \models f\)
- \(\pi \models f_1 \lor f_2\) ⇔ \(\pi \models f_1 \lor \pi \models f_2\)
- \(\pi \models f_1 \land f_2\) ⇔ \(\pi \models f_1 \land \pi \models f_2\)
- \(\pi \models \text{X} f\) ⇔ \(\pi^1 \models f\) \(f\) holds on the path suffix starting from state \(1\)
- \(\pi \models \text{F} f\) ⇔ \(\exists \pi^1 > 0. \pi^1 \models f\) there exists a suffix on which \(f\) holds (\(f\) holds in a state)
- \(\pi \models \text{G} f\) ⇔ \(\forall \pi^1 > 0. \pi^1 \models f\) \(f\) holds on all path suffixes (\(f\) holds in all states)
- \(\pi \models f_1 \text{U} f_2\) ⇔ \(\exists k > 0. \pi^k \models f_2 \land \forall j \leq k. \pi^j \models f_1\)
  \(f_2\) holds on path starting at \(k\) (for some \(k\)), \(f_1\) holds everywhere prior

---

**CTL* syntax and semantics**

Additional path quantifier: \(\text{E}\) there exists a path

Two classes of formulas:
- **state formulas**, evaluated in a state
  - \(f :::= p\) base case: \(p \in \text{AP}\) atomic proposition
  - \(\neg f\) \(\lor f\) \(\land f\) \(f_1, f_2\) state formulas
  - \(\text{E} g\) \(\text{A} g\) path formula
- **path formula**, evaluated over a path
  - \(g :::= f\) base case: \(f\) is state formula
  - \(\neg g\) \(\lor g\) \(\land g\) \(X g\) \(F g\) \(G g\) \(U g\)
  (same rules as LTL, only base case more complex/expressive)

Semantics: same rules as LTL, plus:
- \(s \models \text{E} g\) ⇔ there exists a path \(\pi\) from \(s\) with \(\pi \models g\)
Computation tree logic CTL

defined by Clarke & Emerson (1981)
⇒ Turing Award 2007 with J.Sifakis for model checking
Tradeoff: expressiveness of specifications vs. efficiency of checking
⇒ CTL is subset of CTL*, efficient to check, enough in many cases

CTL is a branching-time logic, like CTL*

CTL quantifies over paths starting from a state
⇒ operators X, F, G, U are immediately preceded by A sau E
⇒ syntax of path formulas simplified, directly using state formulas:

\[ g ::= X f | F f | G f | F_1 U F_2 | F_1 R F_2 \]

Expressiveness: LTL and CTL incomparable (neither includes the other); both less expressive than CTL*

Sample CTL formulas

- EF finish
  It is possible to get to a state in which finish = true.
- AG (send → AF ack)
  Any send is eventually followed by an ack.
- AF AG stable
  On any path, stable is invariant (always holds) after some point
- AG (req → A[req U grant])
  A req stays active until a grant is issued.
- AGAF ready
  On any path ready holds infinitely often.
- AGEF restart
  From any state, it is possible to reach a state labeled restart.

CTL model checking. The operator EU

Idea: backwards traversal from states labeled f2 as long as f1 holds

procedure CheckEU(f1, f2)

\[ T := \{ s | f \in l(s) \} \]

if f holds in s

forall s ∈ T do

\[ (s) := l(s) \cup \{ E[f_1 U f_2] \} \]

if f1|f2 holds, label s

while T ̸= ∅ do

s ∈ T;

never consider s twice

forall s1, R(s1, s) do

if E[f1 U f2] \notin l(s1) \& s \notin l(s1) then

not labeled but f1 holds

l(s1) := l(s1) \cup \{ E[f_1 U f_2] \};

\[ l[f_1 U f_2] \] also holds, label it

T := T \cup \{ s1 \};

s1 is candidate for continuing search

Terminates since S' finite and no labeled state reenters T

Relations between operators

\[ f \land g \equiv (\neg f \lor \neg g) \]
\[ F f \equiv \text{true}_U f \]
\[ G f \equiv \neg F \neg f \]
\[ A f \equiv \neg E \neg f \]
⇒ Operators \( \neg, \land, X, U, E \) and E suffice to express any CTL* formula.

CTL has 2 x 4 = 8 pairs of quantifier x temporal operator:

- AX f \equiv \neg EX f
- EF f \equiv E[true_U f]
- AF f \equiv \neg EG \neg f
- AG f \equiv \neg EF \neg f

⇒ all of them expressible using EX, EU and EG

CTL model checking. The operator EG

Consider only states satisfying f. Traverse backwards starting from strongly connected components (on cycles where f perpetually holds).

procedure CheckEG(f)

restrict to states where f holds

\[ S' := \{ s | f \in l(s) \}; \]

S CC := \{ C | C is nontrivial SCC of S‘ \};

at least one edge

\[ T := U_{l(x) \in S CC} l(x) \in C \];

all states in SCCs are on cycles

forall s ∈ T do

\[ (s) := l(s) \cup \{ EG f \} \];

thus get labeled

while T ̸= ∅ do

\[ \text{choose } s \in T; \]

\[ T := T \setminus \{ s \}; \]

\[ \text{continue from } s \text{ only once} \]

forall s1, s2 ∈ S' \& R(s1, s2) do

for all predecessors of s1 not yet labeled

if EG f \notin l(s1) then

\[ l(s_2) := l(s_1) \cup \{ EG f \}; \]

\[ T := T \setminus \{ s_1 \}; \]

s1 is candidate for continuing search

Terminates; will reach at most every state in S'
Fairness

In practice, we check systems subject to “reasonable” assumptions as:
- a request is not ignored forever (by a scheduler/arbiter)
- communication channels do not continually fail (thus, a message being retransmitted is eventually delivered)

These are properties expressible in CTL*, but not CTL.

⇒ need to extend CTL (semantics) with fairness constraints

Intuitively: decision fairness = if a decision (several transitions from a state) is repeated infinitely often, each branch is eventually taken

Reformulate: each destination state of the decision is eventually reached

Formally: A fairness constraint is a formula in temporal logic.

A path is fair iff the constraint is infinitely often true along the path.

In particular: fairness constraint expressed as set of states ⇒ a fair path passes infinitely often through the set

Model checking CTL with fairness

Augment the Kripke structure $M = (S, S_0, R, L, F)$, with $F \subseteq P(S)$ ($F = \text{set of subsets of states}, \{p_1, \ldots, p_n\}, p_i \subseteq S$)

$$\text{inf}(\pi) \triangleq \{s \mid s = s_i \text{ for infinitely many } i\}$$

(set of states appearing infinitely often on $\pi$)

$\pi$ is a fair path $\Leftrightarrow \forall P \in F. \text{inf}(\pi) \cap P \neq \emptyset$.

($\pi$ passes infinitely often through each set from $F$)

For $\models_F$, (“holds fairly”) replace “path” with “fair path” in semantics

For model checking, define new atomic proposition fair:

$$\text{fair} \in L(s) \Leftrightarrow M, s \models_M \text{EG true}$$

⇒ fair-CTL model checking reduces to CTL for $AP \cup \{\text{fair}\}$

Complexity of model checking algorithms

- CTL model checking: $O(|f| \cdot (|S| + |R|))$
  (linear in size of model and formula)
- CTL with fairness: $O(|f| \cdot (|S| + |R|) \cdot |F|)$
  (linear in size of model and formula)
- LTL: PSPACE-complete
  (different type of algorithm, based on a tableau construction)
- CTL*: like LTL

CTL: usually preferred, because of polynomial (linear!) algorithm

Spin uses LTL: exponential only in size of formula (usually small)

Synchronous and asynchronous composition

Behavior of composed systems emerges from component behavior.

For concurrently executing components: parallel composition:

- synchronous: conjunction (simultaneous transitions)
  $$R(V, V') = R_1(V_1, V'_1) \land R_2(V_2, V'_2) \quad V = V_1 \cup V_2$$

- asynchronous: disjunction (individual transitions)
  $$R(V, V') = R_1(V_1, V'_1) \lor R_2(V_2, V'_2) \lor Eq(V \setminus V_2) \lor Eq(V \setminus V_2)$$

  $$Eq(U) = \bigwedge_{v \in U} (v = v')$$

  – arbitrary interleaving between transitions of components
  – a transition modifies just the variables of one component
  – simultaneous transitions are deemed impossible