Notes on CAFV

Apiros3

First Version : May 20, 2025
Last Update : May 20, 2025

Contents

1 Automata

1.1 Regular Language

- is closed under intersection and complement

1.2 NFA

Definition 1.2.1. *A nondetermininstic finite automation (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where*

- *Q is a finite set of states*
- *Σ is an alphabet*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function*
- *$Q_0 \subseteq Q$ is a set of initial states*
- *$F \subseteq Q$ is a set of accepting states*

Definition 1.2.2. *We define $\mathcal{L}(\mathcal{A})$ to be the set of finite whose run ends in an accept state. (TODO, formally write)*

Definition 1.2.3. *NFAs $\mathcal{A}, \mathcal{A}'$ are equivalent if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A})$.*

Lemma 1.2.4. *A set of finite words $\mathcal{L} \subseteq \Sigma^*$ is a regular language if $\mathcal{L} = \mathcal{L}(\mathcal{A})$ for some finite NFA $\mathcal{A}$.*

Proof. TODO. □

Definition 1.2.5. *We write $\mathcal{A}_1 \otimes \mathcal{A}_2$ for the (canonical) NFA that accepts $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.*

Remark 1.2.6. Given an NFA $\mathcal{A}$, checking $\mathcal{L}(\mathcal{A}) \neq \emptyset$ is equivalent to finding a path that can reach a final state from any state in Q_0, reducing the problem to graph reachability.

1.3 Omega-regular expression

Definition 1.3.1. *An ω-**regular expression** over Σ is of the form*

$$G = E_1(F_1)^\omega + \cdots + E_n(F_n)^\omega$$

*where E_i and F_i are regular expressions with $\epsilon \notin \mathcal{L}(F_i)$. Define $\mathcal{L}_\omega(G) \subseteq \Sigma^\omega$ to be the **language of an ω-regular expression**, defined by*

$$\mathcal{L}_\omega(G) = \mathcal{L}(E_1)\mathcal{L}(F_1)^\omega \cup \cdots \cup \mathcal{L}(E_n)\mathcal{L}(F_n)^\omega$$

where

$$\mathcal{L}(E)^\omega = \{w_1 w_2 w_3 \cdots \mid w_i \in \mathcal{L}(E)\}$$

Definition 1.3.2. *$\mathcal{L} \subseteq \Sigma^\omega$ is an ω-regular language if $\mathcal{L} = \mathcal{L}_\omega(G)$ for some ω-regular expression G.*

1.4 Nondetermininstic Büchi automation

Definition 1.4.1. *A Nondetermininstic Büchi automation (NBA) is a tuple,*

$$\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$$

where

- *Q is a finite set of states*
- *Σ is an alphabet*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function*
- *$Q_0 \subseteq Q$ is the set of initial sets*
- *$F \subseteq Q$ is a set of accept states*

Remark 1.4.2. The setup is the same as NFAs.

Definition 1.4.3. *A **run** of an NBA $\mathcal{A}$ on an infinite word $w = A_0A_1\cdots$ is a sequence of automata states $q_0q_1\cdots$ such that $q_0 \in Q_0$ and $q_i - A_i \to q_{i+1}$ for all $i \geq 0$.*

*An **accepting run** is a run with $q_i \in F$ for infinitely many i.*

The language of $\mathcal{A}$, denoted $\mathcal{L}_\omega(\mathcal{A})$ is the set of all infinite words accepted by $\mathcal{A}$.

Example 1.4.4. Consider the NBA

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop above, "\neg a"] \arrow[r, shift left, "a"] & q_1 \arrow[loop above, "a"] \arrow[l, shift left, "\neg a"]
\end{tikzcd}$$

where q_1 is the only accepting state. Accepted words are exactly the words that accept a infinitely often.

Example 1.4.5. Consider the NBA,

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop above, "\neg a \vee b"] \arrow[r, shift left, "a \wedge \neg b"] & q_1 \arrow[loop above, "\neg b"] \arrow[l, shift left, "b"]
\end{tikzcd}$$

where q_0 is the only accepting state. Then the accepted words are exactly those where b always follows (could be same index) an a.

Lemma 1.4.6. *NBAs are closed under intersection and complementation.*

Lemma 1.4.7. *NBAs are strictly more expressive than DBAs.*

Definition 1.4.8. *An NBA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is **non-blocking** if every symbol is available in every state ($\delta(q, A) \neq \emptyset$ for all $q \in Q$ and $A \in \Sigma$) In particular, every infinite word has a run through $\mathcal{A}$.*

Proposition 1.4.9. *NBAs can always be converted to a non-blocking NBA by adding a 'trap' state that catches every non-used symbol and loops around the trap state.*

Definition 1.4.10. *A generalised nondetermininstic Büchi automata (GNBA) is an NBA with k sets of accept states, where an element from each set must be visited infinitely often (the intersection of states visited infinitely often with each set is nontrivial).*

Lemma 1.4.11. *Given any GNBA $\mathcal{A}$, there is an NBA $\mathcal{A}'$ that accepts the same words.*

Proof. Sketch. Let $F_1, \dots, F_k$ be the set of accepting sets. Given an GNBA, construct an NBA that is k copies of the GNBA such that we move between copies modulo k when accepting sets are reached. Let the accepting states be the accepting states in the k-th copy, with the initial states in the 1st copy. Then we visit accepting states infinitely often if and only if we visit elements of $F_1, \dots, F_k$ each infinitely often. $\square$

2 Graph

2.1 Binary Decision Diagram

Definition 2.1.1. *A binary decision tree is a graphical representation of boolean functions ($f(x_1, \dots, x_n) \mid \{0,1\}^n \to \{0,1\}$) is a perfect binary tree of height n such that*

- *nodes at height $n - i$ are laballed by boolean variables x_{i+1}*
- *the two children have edges laballed 0 (dotted) and 1 (solid)*
- *leafs are labelled with 0 or 1, based on the value of $f(x_1, \dots, x_n)$ with x_i substituted with the edge taken to reach the leaf.*

We can merge isomorphic subtrees to make an DAG (direction is induced) with smaller representation.

Definition 2.1.2. *Fix any total orderig on the variables (the canonical one is induced by the ordering on the tree). We say that the DAG is* ***ordered*** *if for any path along the DAG, variables appear at most once each in the order. It is* ***reduced****, if*

- *Uniqueness: given two non-terminal nodes u, v, if $\mathrm{var}(u) = \mathrm{var}(v), \mathrm{then}(u) = \mathrm{then}(v), \mathrm{else}(u) = \mathrm{else}(v)$, then $u = v$, where* var *is the variable on the node,* then *and* else *are the nodes reached by following* 1 *and* 0 *respectively.*
- *Non-redundant: for any non-terminal node u, $\mathrm{then}(u) \neq \mathrm{else}(u)$.*
- *Terminal nodes are merged*

If the DAG is reduced and ordered, it is called a ***binary decision diagrams (BDD)****.*

Remark 2.1.3. We can reduce heuristically reduce towards a BDD by the following techniques:

- Merge isomorphic nodes (including terminal ones)
- Remove redundant nodes (with identical children)

Theorem 2.1.4. *Given a fixed ordering on the variables, for any propositional formula ϕ, there exists a unique BDD equivalent to ϕ.*

Proof. OoSN. $\square$

Corollary 2.1.5. *Given a fixed ordering on the variables, two boolean functions are equivalent if and only if the reduced, ordered BDDs are isomorphic.*

Proof. Follows immediately from the previous theorem. □

Proposition 2.1.6. *Given a BDD, satisfiability checking, tautology checking are both constant time problems.*

Proof. A BDD represents a satisfiable boolean function if there are any edges into 1 (or the 1 leaf exists). It is a tautology if there are no edges into 0 (or the 0 leaf does not exist). □

Proposition 2.1.7. *Equality checking of BDDs can be done in linear time.*

Proof. OoSN. □

Example 2.1.8. Consider the DNF,

$$f = (x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)$$

The interleaved ordering $x_1 < y_1 < \cdots < x_n < y_n$ gives a BDD of size $2n + 2$, whereas the ordering $x_1 < \cdots < x_n < y_1 < \cdots < y_n$ gives a BDD of size 2^{n+1}.

Theorem 2.1.9. *Finding the ordering that gives the minimal BDD size is NP-complete.*

Proof. OoSN. □

Given BDDs (with some fixed ordering), that represents a boolean function, we can give BDDs that represent negation, conjunction, and disjunction. Negation is simply replacing the children out of each node with one-another.

Example 2.1.10. Given BDDs that represent boolean functions A and B, we can give a BDD that represents $A \vee B$ (and similar for conjunction) in $O(|A||B|)$ time. The algorithm is as follows:

1. Start with the pair of root nodes in A and B. Recursively do the following:

2. Pick the smaller variable of the pair, and draw an edge to the pair that corresponds to substitution of 0 and 1. Recursively apply the process on the new pair.

The label on terminal nodes (u, v) is simply $\text{var}(u) \vee \text{var}(v)$ (or $\wedge$ if conjunction). The resulting BDD needs to be reduced, but this can be done as part of the recursive operation by implementing the reduction rules in a bottom-up fashion.

When implementing BDDs, one can do so efficiently (memory-wise) by making a multi-rooted BDD such that there are no duplicate BDD subtrees accross multiple BDDs. Whenever a new node is created, check for existence first (if exists, attach to that). BDD equality becomes trivial by a simple pointer comparison.

Thus, set of states can be implemented by bit vectors and hashing, whereas transition relations are represented by a sparse adjacency matrix.

3 Logic

To add: up to logic, $\sigma \not\models \psi$ is equivalent $\sigma \models \neg\psi$

3.1 Propositional Logic

Definition 3.1.1. *Propositional logic formulas are spanned by*

$$\phi ::= \text{true} \mid \text{false} \mid a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$$

where $a \in \mathrm{AP}$ is an atomic proposition.

Often we give a slightly more minimal set of grammars,

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi$$

As then we have the logical equivalences

- $\text{false} \equiv \neg\text{true}$
- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \to \phi_2 \equiv \neg\phi_1 \wedge \phi_2$
- $\phi_1 \iff \phi_2 \equiv (\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1)$
- $\phi_1 \oplus \phi_2 \equiv (\phi_1 \wedge \neg\phi_2) \vee (\neg\phi_1 \wedge \phi_2)$

3.1.1 Normal Forms and Satisfiability

Definition 3.1.2. *Let ϕ be a propositional logic formula over boolean variables $x_1, \dots, x_n$. We say that ϕ is **satisfiable** if there exists a valuation of $x_1, \dots, x_n$ such that $\phi(x_1, \dots, x_n)$ evaluates to true.*

Theorem 3.1.3 (Cook's Theorem)**.** *Checking satisfiability is NP-complete.*

Proof. todo. □

Definition 3.1.4. *Let ϕ is a **tautology** if $\phi(x_1, \dots, x_n)$ evaluates to true for every valuation of $x_1, \dots, x_n$.*

Theorem 3.1.5. *Tautology checking is co-NP-complete.*

Definition 3.1.6. *A formula ϕ is in **conjunctive normal form** (CNF) if $\phi = \bigwedge \bigvee \ell_{ij}$ where ℓ_{ij} is a literal of the form x_k or $\neg x_k$.*

Lemma 3.1.7. *Any boolean formula is equivalent to a CNF-formula.*

Proof. Use de Morgan, double negation, and distributivity. □

Proposition 3.1.8. *Tautology checking of a formula in CNF is in P.*

Proof. Tautology checking for a ϕ in CNF is equivalent to tautology checking each conjunctive clause. One can tautology check each clause by seeing if for every literal, it's opposite literal is also in it. □

Definition 3.1.9. *A formula ϕ is in **disjunctive normal form** (DNF) if $\phi = \bigvee \bigwedge \ell_{ij}$ where ℓ_{ij} is a literal of the form x_k or $\neg x_k$.*

Lemma 3.1.10. *Any boolean formula is equivalent to a DNF-formula.*

Proof. Use de Morgan, double negation, and distributivity. □

Proposition 3.1.11. *Satisfiability checking of a formula in DNF is in P.*

Proof. SAT checking reduces to seeing if any clause is satisfied. For every clause, we check that no literal has both a positive and negative occurence, and this is equivalent to SAT checking. □

Proposition 3.1.12. *Tautology checking of a formula in DNF is in co-NP-complete.*

Proof. OoSN. □

Example 3.1.13. For a DNF size $O(n)$, the equivalent CNF can be size $\Omega(2^n)$. Consider $(x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)$. It's CNF needs to cover every 2^n possibilities of satisfiability.

Definition 3.1.14. *A formula ϕ is in **if-then-else normal form** (INF), if*

$$\phi ::= 1 \mid 0 \mid x \to \phi, \phi$$

where x is a boolean variable and $x \to \phi_1, \phi_2 \equiv (x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$.

Theorem 3.1.15 (Shannon's expansion theorem)**.** *For every Boolean formula ϕ and variable x, $\phi \equiv x \to \phi[1/x], \phi[0/x]$.*

Proof. OoSN. □

Corollary 3.1.16. *Any Boolean formula is equivalent to one in INF.*

Proof. Immediate consequence of Shannon's expansion theorem, after substitution into all variables it should reduce to a 1 or 0.

Alternatively, one may simply prove this through structural induction. □

3.2 Linear Temporal Logic

Definition 3.2.1. *Linear Temporal Logic (LTL) formulas ψ is defined by the grammar*

$$\psi ::= \text{true} \mid a \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi \cup \psi$$

The temporal operators mean the following:

- *$\bigcirc\psi$ means that ψ is true in the next state*
- *$\psi_1 \cup \psi_2$ means that ψ_2 is eventually true and ψ_1 is true until then*

Notation 3.2.2. Given a word $\sigma \in (2^{\text{AP}})^\omega$, we can write $\sigma = A_0A_1A_2\cdots$. We write $\sigma[j] = A_j$ and $\sigma[j...] = A_jA_{j+1}A_{j+2}\cdots$

We write semantics as follows:

- $\sigma \models \text{true}$ is always true
- $\sigma \models a$ if and only if $a \in \sigma[0]$
- $\sigma \models \psi_1 \wedge \psi_2$ if and only if $\sigma \models \psi_1$ and $\sigma \models \psi_2$
- $\sigma \models \neg\psi$ if and only if $\sigma \models \psi$

- $\sigma \models \bigcirc\psi$ if and only if $\sigma[1...] \models \psi$
- $\sigma \models \psi_1 \cup \psi_2$ if and only if there exists $k \geq 0$ such that $\sigma[k...] \models \psi_2$ and for all $i < k$, $\sigma[i...] \models \psi_1$

Definition 3.2.3. *LTL formulae ψ_1, ψ_2 are* ***equivalent****, written $\psi_1 \equiv \psi_2$ if they are satisfied by the exact same ω-word (in 2^{AP}). That is, $\sigma \models \psi_1$ if and only if $\sigma \models \psi_2$.*

Definition 3.2.4. *We define* $\mathrm{Words}(\psi) = \{\sigma \in (2^{\mathrm{AP}})^\omega \mid \sigma \models \psi\}$.

Remark 3.2.5. By simple rewriting, we have that $\psi_1 \equiv \psi_2$ if and only if $\mathrm{Words}(\psi_1) = \mathrm{Words}(\psi_2)$.

Lemma 3.2.6. *$\psi_1 \equiv \psi_2$ if and only if for any LTS M, $M \models \psi_1 \iff M \models \psi_2$*

Proof. OoSP. (Not trivial, check ltr) (apparently standard automata-theoretic filtration argument??) $\square$

The operators are powerful enough to talk about additional operators:

- Eventually ψ : $\Diamond\psi \equiv \mathrm{true} \cup \psi$
- Always ψ : $\Box\psi \equiv \neg(\mathrm{true} \cup \neg\psi)$

Notation 3.2.7. We have alternative syntax:

- $\bigcirc a \equiv X\ a$
- $\Diamond a \equiv F\ a$
- $\Box a \equiv G\ a$

Lemma 3.2.8 (Distributivity)**.** *We have the following distributivity laws*

$$\bigcirc(\psi_1 \vee \psi_2) \equiv (\bigcirc\psi_1) \vee (\bigcirc\psi_2), \quad \bigcirc(\psi_1 \wedge \psi_2) \equiv (\bigcirc\psi_1) \wedge (\bigcirc\psi_2), \quad \bigcirc(\psi_1 \cup \psi_2) \equiv (\bigcirc\psi_1) \cup (\bigcirc\psi_2)$$
$$\Box(\psi_1 \wedge \psi_2) \equiv \Box\psi_1 \wedge \Box\psi_2, \quad \Diamond(\psi_1 \vee \psi_2) \equiv \Diamond\psi_1 \vee \Box\psi_2,$$
$$\psi_1 \cup (\psi_2 \vee \psi_3) \equiv (\psi_1 \vee \psi_2) \cup (\psi_1 \vee \psi_3), \quad (\psi_1 \wedge \psi_2) \cup \psi_3 \equiv (\psi_1 \cup \psi_3) \wedge (\psi_2 \cup \psi_3)$$

Proof. Straightforward. $\square$

Lemma 3.2.9 (Duality / Negation Propagation)**.** *We have the following dualities*

$$\neg \bigcirc \psi \equiv \bigcirc\neg\psi \quad \neg\Box\psi \equiv \Diamond\neg\psi \quad \neg\Diamond\psi \equiv \Box\neg\psi$$

Proof. Straightforward. $\square$

Lemma 3.2.10 (Other Properties)**.** *We also note idempotency and expansion laws:*

$$\Box\Box\psi \equiv \Box\psi \quad \Diamond\Diamond\psi \equiv \Diamond\psi$$
$$\psi_1 \cup \psi_2 \equiv \psi_1 \cup (\psi_1 \cup \psi_2) \quad \psi_1 \cup \psi_2 \equiv \psi_1 \vee (\psi_1 \wedge \bigcirc(\psi_1 \cup \psi_2))$$
$$\Box\psi \equiv \psi \wedge \bigcirc\Box\psi \quad \Diamond\psi \equiv \psi \vee \bigcirc\Diamond\psi$$

Proof. Straightforward. (At least in the cafv course, the standard equivalence we use is $\Diamond a \equiv \neg\Box\neg a$ and $\Box a \equiv \neg\Diamond\neg a$) $\square$

Remark 3.2.11. Proofs for all of the above are straightforward, by using the fact that $A \equiv B$ if and only if for all σ, $\sigma \models A \iff \sigma \models B$, or using known equivalences.

Example 3.2.12. We will show:

$$\Box\Diamond a \wedge \Box\Diamond b \not\equiv \Box\Diamond(a \wedge b)$$

As the word $\{a\}\{b\}\{a\}\{b\}\cdots$ is satisfied only by the left.

3.3 Computation Tree Logic

If LTL was based on a time-linear system, CTL is the version which deals with branching to different paths over time. Intuitively, linear time paths have a chosen path to model over, whereas computation trees have branching that can be ‘chosen’ afterwards.

Definition 3.3.1. *A computation tree logic (CTL) formula is a* ***state formula*** ϕ *given by*

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall\psi \mid \exists\psi$$

where

$$\psi ::= \bigcirc\phi \mid \phi \cup \phi \mid \Diamond\phi \mid \Box\phi$$

Note that temporal operators are paired with quantifiers.

Notation 3.3.2. We can write $\forall\psi \equiv A\ \psi$ and $\exists\psi \equiv E\ \psi$.

Definition 3.3.3. *The* ***release operator*** $\phi_1 R \phi_2$ *corresponds to ‘*ϕ_2 *always holds, but is no longer required after* ϕ_1 *holds’. That is,* $\pi \models \phi_1 R \phi_2$ *if either* $\pi[j] \models \phi_2$ *for all* $j \geq 0$*, or there exists a* $i \geq 0$ *such that* $\pi[i] \models \phi_1$ *and* $\pi[k] \models \phi_2$ *for all* $k \leq i$*.*

Proposition 3.3.4 (Release is the dual of until)**.** *We have that* $\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 \cup \neg\phi_2)$ *and* $\neg(\phi_1 \cup \phi_2) \equiv \neg\phi_1 R \neg\phi_2$*.*

3.3.1 Positive Normal Form

Definition 3.3.5. *A CTL formula is in* ***positive normal form (PNF)*** *if negation is applied only to atomic propositions.*

Lemma 3.3.6. *Any CTL formula can be converted to an equivalent formula in PNF.*

Proof. We push negation inside inductively by using duality. □

4 Labelled Transition Systems

4.1 Basic Definitions

States represent possible configurations of systems, whereas transitions represents possible ways the system can evolve.

Definition 4.1.1. *A* ***labelled transition system (LTS)*** *is a tuple* $(S, \text{Act}, \to, I, \text{AP}, L)$ *where*

- S *is a set of states (state space)*
- Act *is a set of actions*
- $\to \subseteq S \times \text{Act} \times S$ *is a transition relation*
- $I \subseteq S$ *is a set of initial states*
- AP *is a set of atomic propositions*
- $L : S \to 2^{\text{AP}}$ *is a labelling function*

Essentially is a directed graph where vertices represent states and edges represent transitions.

States are represented using atomic propositions in AP, which represent facts about each state. Transitions are labelled with actions.

Definition 4.1.2. *An LTS is **finite** if S, Act, AP are all finite.*

Notation 4.1.3. We write $s - \alpha \to s'$ if $(s, \alpha, s') \in \to$ and $s \to s'$ if $s - \alpha \to s'$ for some α

Definition 4.1.4. *We define $\mathrm{Post}(s, \alpha) = \{s' \in S \mid s - \alpha \to s'\}$ and $\mathrm{Post}(s) = \bigcup_{\alpha \in \mathrm{Act}} \mathrm{Post}(s, \alpha)$.*

Similarly, $\mathrm{Pre}(s, \alpha) = \{s' \in S \mid s' - \alpha \to s\}$ and $\mathrm{Pre}(s) = \bigcup_{\alpha \in \mathrm{Act}} \mathrm{Pre}(s, \alpha)$.

*A state s is called **terminal** if $\mathrm{Post}(s) = \emptyset$.*

Terminal states often represent termination of a program or error / undesired behavior like deadlock.

Definition 4.1.5. *A **path** is an alternating sequence $s_0\alpha_0 s_1 \alpha_1 s_2 \alpha_2 \cdots$ such that $s_1 - \alpha_i \to s_{i+1}$.*

*A **finite path** or a path fragment is a finite prefix of a path, ending in a state.*

Definition 4.1.6. *A state s' is **reachable** from s if there exists a finite path from s to s'. s is called a **reachable state** if it is reachable from some $s_0 \in I$.*

Notation 4.1.7. We write A^ω to be $A(A^\omega)$.

Remark 4.1.8. $\mathrm{Post}^*(s)$ is the set of states reachable from s, and $\mathrm{Pre}^*(s)$ is the set of states that can reach s.

As notation, we write $\mathrm{Post}^*(C) = \bigcup_{s \in C} \mathrm{Post}^*(s)$ for a set $C \subseteq S$ and Similarly for Pre.

Given an LTS M, we write $\mathrm{Reach}(M) = \mathrm{Post}^*(I)$

Example 4.1.9. A program can be modelled using an LTS by a set of tuples that represent the location (in code) and what the variables are.

Remark 4.1.10. LTS exhibits nondeterminism, as we can have an unknown system environment (like user input), abstraction (omitting detail like probability of success), underspecification, or concurrency. This is seen as nondeterminism over the choice of transition in each state and the choice of initial state.

Definition 4.1.11. *The **distance** $\delta(s, t)$ from a state s to state t in LTS M is the length of the shortest path between them. That is,*

$$\delta(s, t) = \min\{n \mid \exists s_0, \ldots, s_n, s_0 = s \wedge s_n = t \wedge \forall 0 \leq i < n, s_i \to s_{i+1}\}$$

where $\min \emptyset = \infty$.

Definition 4.1.12. *The **diameter** $d(M)$ of a LTS M is the maximal length of a path to any state. That is,*

$$d(M) = \max\{d(t) \mid d(t) \neq \infty \wedge t \in S\}$$

where

$$d(t) = \min\{\delta(s, t) \mid s \in I\}$$

Definition 4.1.13. *The **recurrence diameter** $rd(M)$ of a LTS M is the longest loop-free path. That is,*

$$rd(M) = \max\{n \mid \exists s_0, \ldots, s_n, s_0 \in I \wedge \forall 0 \leq i < n, s_i \to s_{i+1} \wedge \forall 0 \leq i < n, \forall i < j \leq n, s_i \neq s_j\}$$

Note that each value can be computed symbolically with a SAT solver.

Proposition 4.1.14. *For any LTS M, $rd(M) \geq d(M)$.*

Proof. Straightforward. Any minimal path to a state us a loop-free path. □

4.1.1 Parallel Composition

Definition 4.1.15 (Composition of LTS). *Let M_1 and M_2 be two LTSs, where $M_i = (S_i, \mathrm{Act}_i, \to_i, I_i, \mathrm{AP}_i, L_i)$. We define interleaving $M_1|||M_2$ as the LTS*

$$M_1|||M_2 := (S_1 \times S_2, \mathrm{Act}_1 \cup \mathrm{Act}_2, \to, I_1 \times I_2, \mathrm{AP}_1 \cup \mathrm{AP}_2, L)$$

where $L((s_1, s_2)) = L(s_1) \cup L(s_2)$ for any $s_1 \in S_1$ and $s_2 \in S_2$, and $\to$ is defined such that

$$\frac{s_1 - \alpha \to_1 s_1'}{(s_1, s_2) - \alpha \to (s_1', s_2)} \qquad \frac{s_2 - \alpha \to_2 s_2'}{(s_1, s_2) - \alpha \to (s_1, s_2')}$$

Example 4.1.16. Consider parallel composition over programs that have shared access to some variable,

$$[x := x + 1]|||[y := 2 * x]$$

Nondeterminism models competition between these variables (with two branches representing the different cases of who won)

Definition 4.1.17. ***Synchronisation*** *between parallel components M_1, M_2 is $M_1||_H M_2$ for a set $H \subseteq \mathrm{Act}$ of handshake actions such that synchronisation happens only on those actions. Explicitly,*

$$M_1||_H M_2 = (S_1 \times S_2, \mathrm{Act}_1 \cup \mathrm{Act}_2, \to, I_1 \times I_2, \mathrm{AP}_1 \cup \mathrm{AP}_2, L)$$

where $\to$ is defined as

$$\frac{s_1 - \alpha \to_1 s_1' \land s_2 - \alpha \to_2 s_2'}{(s_1, s_2) - \alpha \to (s_1', s_2')}\ \alpha \in H \qquad \frac{s_1 - \alpha \to_1 s_1'}{(s_1, s_2) - \alpha \to (s_1', s_2')}\ \alpha \notin H \qquad \frac{s_2 - \alpha \to_2 s_2'}{(s_1, s_2) - \alpha \to (s_1', s_2')}\ \alpha \notin H$$

4.2 Linear Time Properties

For this section we assume LTSs are finite and have no terminal states (such that all maximal paths are infinite).

Definition 4.2.1. *We define the **trace** of a path $\pi = s_0 s_1 \cdots$ to be the sequence of atomic propositions true in each state, written $\mathrm{trace}(\pi) = L(s_0)L(s_1)\cdots$*

Notation 4.2.2. We write $\mathrm{Paths}(M)$ to be the set of all paths starting from an initial state in I. We also write $\mathrm{Traces}(M)$ for the set of all traces of those paths.

Definition 4.2.3. *A **linear-time (LT) property** is a subset of $(2^{\mathrm{AP}})^\omega$*

Definition 4.2.4. *A satisfaction $M \models P$ of an LT property P by an LTS M, or M satisfies P when $\mathrm{Traces}(M) \subseteq P$.*

Example 4.2.5. The property P: “the traffic lights never both show green_1 and green_2 simultaneously” is written as

$$P = \{A_0 A_1 A_2 \cdots \in (2^{\mathrm{AP}})^\omega \mid A_j \notin \{\mathrm{green}_1, \mathrm{green}_2\} \text{for all } j \geq 0\}$$

Definition 4.2.6. *M and M' are **trace equivalent** if $\mathrm{Traces}(M) = \mathrm{Traces}(M')$ Similarly, M is a trace inclusion of M' if $\mathrm{Traces}(M) \subseteq \mathrm{Traces}(M')$.*

Proposition 4.2.7. *We have*

- *M and M' are trace equivalent if and only if for any LT property P, $M' \models P \iff M \models P$*
- *M is a trace inclusion of M' if and only if for any LT property P, $M' \models P \implies M \models P$.*

Proof. Straightforward. □

4.2.1 Classes of Linear Time Properties

Definition 4.2.8. $P_{\text{inv}} \subseteq (2^{\text{AP}})^\omega$ *is an **invariant** if there is a prositional logic formula* ϕ *such that*

$$P_{\text{inv}} = \{A_0 A_1 A_2 \cdots \in (2^{\text{AP}})^\omega \mid A_j \models \phi \textit{ for all } j \geq 0\}$$

It is a condition about states that must always be true, and hence can be checked in each state separately.

Checking invariants can be done via reachability. Specifically, we have

Proposition 4.2.9. *The following are equivalent:*

- $M \models P$
- *for all* $\pi \in \text{Paths}(M)$*,* $\text{Trace}(\pi) \in P$
- *for all* $\pi \in \text{Paths}(M)$*,* $\forall s \in \pi, L(s) \models \phi$
- *for all* $s \in \text{Reach}(M), L(s) \models \phi$

Proof. Straightforward. □

Remark 4.2.10. As LTSs are finite, we can check reachability and check that $L(s) \models \phi$ to determine if $M \models P$ in finite time.

Definition 4.2.11. $P_{\text{safe}} \subseteq (2^{\text{AP}})^\omega$ *is a **safety property** if for all words* $\sigma \in (2^{\text{AP}})^\omega \setminus P_{\text{safe}}$*, there is a finite prefix* σ' *of* σ *such that*

$$P_{\text{safe}} \cap \{\sigma'' \in (2^{\text{AP}})^\omega \mid \sigma' \textit{ is a prefix of } \sigma''\} = \emptyset$$

In particular, it is a property such that if there is a violation, there is a finite prefix (evidence) such that every infinite path extending the finite prefix does not satisfy P_{safe}.

Remark 4.2.12. Invariants are safety properties, but the converse is not true. In the former, given that the invariant is established by ϕ, then the set of 'bad' prefixes are of the form $A_0 A_1 \cdots A_n$ such that $A_i \not\models \phi$ (and is a word that is a prefix of an element of a path).

The latter is clearly not true, by considering sequences that depend on where it came from (e.g., green_1 always appears before green_2).

Definition 4.2.13. *Given a trace* $\sigma \in (2^{\text{AP}})^\omega$*, define*

$$\text{pref}(\sigma) = \{\sigma' \in (2^{\text{AP}})^* \mid \sigma' \textit{ is a finite prefix of } \sigma\}$$

and standard overloading to a linear time property P*. Also, take*

$$\text{closure}(P) = \{\sigma \in (2^{\text{AP}})^\omega \mid \text{pref}(\sigma) \subseteq \text{pref}(P)\}$$

Proposition 4.2.14. *Given a LT property* P*,* P *is a safety property if and only if* $\text{closure}(P) = P$*.*

Proof. Intuition: Things in the closure cannot act as 'evidence' against nonmembership. We always have $\text{closure}(P) \supseteq P$, and if we have $\sigma \in \text{closure}(P) \setminus P$, by safety, we have a finite prefix of σ, say σ' that acts as evidence. Now, every extension of σ' is not in P, which contradicts the fact $\sigma' \in \text{pref}(P)$.

If P is not safe, the element which contradicts the safety property is exactly what we use to show strict inclusion. □

Definition 4.2.15. $P_{\text{live}} \subseteq (2^{\text{AP}})^\omega$ *is a **liveness property** if for all finite word* $\sigma \in (2^{\text{AP}})^*$*, there exists an infinite word* $\sigma' \in (2^{\text{AP}})^\omega$ *such that* $\sigma\sigma' \in P_{\text{live}}$*.*

Proposition 4.2.16. P *is live if and only if the* $\text{pref}(P_{\text{live}}) = (2^{\text{AP}})^*$

Proof. Clear. □

4.3 Semantics

4.3.1 Over Propositional Logic

Definition 4.3.1. *Given a state s on a LTS M, we write $s \models \phi$ if the model that assigns only elements of $L(S)$ to be true derives ϕ*

4.3.2 Over LTL

Definition 4.3.2. *Given an LTS M, it satisfies an LTL formula ψ if for all paths, its trace satisfies ψ. That is, $M \models \psi$ if for all $\pi \in \mathrm{Paths}(M)$, $\mathrm{trace}(\pi) \models \psi$.*

Then $M \models \psi$ if and only if $\mathrm{Traces}(M) \subseteq \mathrm{Words}(\psi)$

Lemma 4.3.3. *LTL can represent invariants, safety properties, and liveness properties. Moreover, every invariant can be represented as $\Box\phi$ for some propositional formula ϕ*

Proof. OoSN. □

Example 4.3.4. $\Box(\text{receive} \to \bigcirc\text{ack})$ is a safety property that says "ack always immediately follows receive". $\Diamond\phi$, $\Box\Diamond\phi$ are both liveness properties.

Remark 4.3.5. $M \models \neg\psi$ implies $M \not\models \psi$ (assuming nonempty trace) but not the other way around. This is a direct consequence of the fact that to not model only requires evidence of the negation by one trace, where as a model of the negation requires that it is not true in every trace.

LTLs can model existence well in the sense that it can verify $M \not\models A$ by exhibiting a trace that satisfies $\neg A$. However, it cannot verify things that talk about 'every' execution.

Proposition 4.3.6. *$M \models \psi$ if and only if $\mathrm{Traces}(M) \cap \mathrm{Words}(\neg\psi) = \emptyset$*

Proof. Clear. □

Remark 4.3.7. This equivalence allows the generation of counterexamples in LTL.

4.3.3 Over CTL

Definition 4.3.8. *Given a state s of an LTS $M = (S, \mathrm{Act}, \to, I, \mathrm{AP}, L)$, we have for states,*

- *$s \models \mathrm{true}$ is always true*
- *$s \models a$ if and only if $a \in L(s)$*
- *$s \models \phi_1 \wedge \psi_2$ if and only if $s \models \phi_1$ and $s \models \phi_2$*
- *$s \models \neg\phi$ if and only if $s \not\models \phi$*
- *$s \models \forall\psi$ if and only if forall $\pi \in \mathrm{Paths}(s)$, $\pi \models \psi$*
- *$s \models \exists\psi$ if and only if for some $\pi \in \mathrm{Paths}(s)$, $\pi \models \psi$*

and for paths,

- *$\pi \models \bigcirc\phi$ if and only if $\pi[1] \models \phi$*
- *$\pi \models \phi_1 \cup \phi_2$ if and only if $\exists k \geq 0$ such that $\pi[k] \models \phi_2$ and for all $i < k$, $\pi[i] \models \phi_1$*

Then, we write $M \models \phi$ if for all $s_0 \in I$, we have $s_0 \models \phi$

Definition 4.3.9. *ϕ_1 and ϕ_2 are **equivalent**, written $\phi_1 \equiv \phi_2$ if for any state s of any LTS M, $s \models \phi_1 \iff s \models \phi_2$*

Lemma 4.3.10 (Path quantifier duality)**.** *We have*

$$\forall\psi \equiv \neg\exists\neg\psi \quad \exists\psi \equiv \neg\forall\neg\psi$$

4.4 CTL vs LTL

Definition 4.4.1. *A CTL formula ϕ and LTL formula ψ are equivalent if for any LTS M,*

$$M \models \phi \iff M \models \psi$$

Lemma 4.4.2. *There are formulae in LTL that cannot be expressed in CTL. Specifically, $\Diamond\Box a$ has no equivalent formula in CTL.*

Example 4.4.3. We have $\forall\Diamond\forall\Box a \not\equiv \Diamond\Box a$. Consider the LTS,

$$\begin{tikzcd} {} \arrow[r] & s_0 \arrow[loop above] \arrow[r] & s_1 \arrow[r] & s_2 \arrow[loop above] \end{tikzcd}$$

where s_0 and s_2 have labels $\{a\}$. Every path of this LTS is of the form $s_0 s_0 \cdots$ or $s_0 s_0 \cdots s_0 s_1 s_2 \cdots$, both of which satisfy $\Diamond\Box a$. On the other hand, considering the path $s_0 s_0 \cdots$, we clearly don't have $\Diamond\forall\Box a$, as the secondary path may fall to s_2 through s_1.

Lemma 4.4.4. *There are formulae in CTL that cannot be expressed in LTL. Speficially, $\forall\Box\exists\Diamond a$ cannot be expressed in a LTL.*

Proof. Suppose there exists a LTL formula ψ equivalent to $\forall\Box\exists\Diamond a$. Consider the LTS M defined by,

$$\begin{tikzcd} {} \arrow[r] & s_0 \arrow[loop above] \arrow[r] & s_1 \arrow[loop above] \end{tikzcd}$$

where the label for s_1 is $\{a\}$ and empty otherwise. Note that we have $M \models \forall\Box\exists\Diamond a$. By equivalence, we have $M \models \psi$, such that $\text{Traces}(M) \subseteq \text{Words}(\psi)$.

Now consider the LTS M' defined by

$$\begin{tikzcd} {} \arrow[r] & s_0 \arrow[loop above] \end{tikzcd}$$

with labels to the emptyset such that $\text{Traces}(M') \subseteq \text{Traces}(M)$. We therefore have $M' \models \psi$. However, $M' \not\models \forall\Box\exists\Diamond a$, a contradiction. ☐

Remark 4.4.5. Expressiveness of CTL and LTL are incomparable. The key difference is that CTLs is a branching time, state-based logic, whereas LTLs are a linear-time, path-based model.

Theorem 4.4.6. *Let ϕ be a CTL and ψ be the LTL obtained by removing the quantifiers from ϕ. Then $\phi \equiv \psi$ or there exists no LTL formula equivalent to ϕ.*

Proof. Check PoM Thm 6.18, no proof given. ☐

4.5 CTL*

Definition 4.5.1. *CTL* formulae are ϕ defined on states formulas*

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall\psi \mid \exists\psi$$

where path formulas are

$$\psi ::= \phi \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi \cup \psi \mid \Diamond\psi \mid \Box\psi$$

This is a superset of CTL and LTL, where we can have nested temporal operators.

Definition 4.5.2. *We give semantics on states s of an LTS $M = (S, \text{Act}, \to, I, \text{AP}, L)$ with*

- $s \models \text{true}$ *is always true*
- $s \models a$ *if and only if* $a \in L(s)$
- $s \models \phi_1 \wedge \phi_2$ *if and only if* $s \models \phi_1$ *and* $s \models \phi_2$
- $s \models \neg\phi$ *if and only if* $s \not\models \phi$
- $s \models \forall\psi$ *if and only if for all* $\pi \in \text{Path}(s)$, $\pi \models \psi$
- $s \models \exists\psi$ *if and only if there exists some* $\pi \in \text{Path}(s)$ *such that* $\pi \models \psi$

and for a path π,

- $\pi \models \phi$ *if and only if* $\pi[0] \models \phi$
- $\pi \models \psi_1 \wedge \psi_2$ *if and only if* $\pi \models \psi_1$ *and* $\pi \models \psi_2$
- $\pi \models \neg\psi$ *if and only if* $\pi \not\models \psi$
- $\pi \models \bigcirc\psi$ *if and only if* $\pi[1...] \models \psi$
- $\pi \models \psi \cup \psi_2$ *if and only if there exists a* $k \geq 0$ *such that* $\pi[k...] \models \psi_2$ *and for all* $i < k$, $\pi[i...] \models \psi_1$.

Definition 4.5.3. *CTL* formulae ϕ_1 and ϕ_2 are equivalent if for all LTS M, we have*

$$M \models \phi_1 \iff M \models \phi_2$$

Example 4.5.4. Given an LTS M, we can have $M \not\models \phi$ and $M \not\models \neg\phi$. Consider $\phi = \exists\Box a$ such that $\neg\phi = \neg\exists\Box a \equiv \forall\Diamond\neg a$.

$$\begin{tikzcd}
{} \arrow[r] & s_0 \arrow[r] \arrow[loop above] & s_1 \arrow[loop above] & s_2 \arrow[l] & {} \arrow[l]
\end{tikzcd}$$

where labels on s_0 and s_2 are $\{a\}$ and empty otherwise. The consequence is due to the propositions needing to be satisfied by all initial states. ϕ is not satisfied by the path starting from s_2, and $\neg\phi$ is not satisfied by the path starting from s_0.

Definition 4.5.5. *Any two states s_1 and s_2 are* ***CTL* equivalent****, written $s_1 \equiv_{\text{CTL}^*} s_2$, if $s_1 \models \phi \iff s_2 \models \phi$ for all CTL* formulas ϕ. Similarly, two LTSs M_1 and M_2 are CTL* equivalent, written $M_1 \equiv_{\text{CTL}^*} M_2$*

Remark 4.5.6. This notion then induces LTL-equivalences and CTL-equivalences.

5 Model Checking

5.1 CTL

Definition 5.1.1. *A **witness** for a CTL formula $\exists\psi$ on a LTS M is a sufficiently long prefix of a path π of M with $\pi \models \psi$ such that it shows how ψ can be true.*

*A **counterexample** for a CTL formula $\forall\psi$ on a LTS M is a sufficiently long prefix of a path π of M with $\pi \not\models \psi$ (a witness for $\neg\psi$).*

Definition 5.1.2. *We define* $\mathrm{Sat}(\phi)$*, called the **satisfaction set** for the CTL formula ϕ as*

$$\mathrm{Sat}(\phi) := \{s \in S \mid s \models \phi\}$$

Definition 5.1.3. *Existential Normal Form (ENF) for a CTL is a formula with no $\forall$ and no $\exists\Diamond$.*

Lemma 5.1.4. *Every formula has an equivalent formula in ENF.*

Proof. Sketch. Every $\forall$ can be removed using the path quantifier duality, $\forall\psi \equiv \neg\exists\neg\psi$ (where we use duality again on the temporal operator immediately.)

We also have that $\Diamond$ and $\Box$ are derived operators (or simply take the dual), and that,

- $\exists\Diamond\phi \equiv \exists(\mathrm{true} \cup \phi)$
- $\forall \bigcirc \phi \equiv \neg\exists \bigcirc \neg\phi$
- $\forall(\phi_1 \cup \phi_2) \equiv \neg\exists(\neg\phi_2 \cup (\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\exists(\Box\neg\phi_2)$

□

Now, to check whether $M \models \phi$, we want to see if $s \models \phi$ for all initial states $s \in I$. It is therefore sufficient to check that $I \subseteq \mathrm{Sat}(\phi)$. We can compute $\mathrm{Sat}(\phi)$ recursively on ϕ. Writing ϕ in ENF, we have

- $\mathrm{Sat}(\mathrm{true}) = S$
- $\mathrm{Sat}(a) = \{s \in S \mid a \in L(s)\}$
- $\mathrm{Sat}(\phi_1 \wedge \phi_2) = \mathrm{Sat}(\phi_1) \cap \mathrm{Sat}(\phi_2)$
- $\mathrm{Sat}(\neg\phi) = S \setminus \mathrm{Sat}(\phi)$
- $\mathrm{Sat}(\exists \bigcirc \phi) = \{s \in S \mid \mathrm{Post}(s) \cap \mathrm{Sat}(\phi) \neq \emptyset\}$
- $\mathrm{Sat}(\exists(\phi_1 \cup \phi_2)) = \mathrm{CheckExistsUntil}(\mathrm{Sat}(\phi_1), \mathrm{Sat}(\phi_2))$
- $\mathrm{Sat}(\exists\Box\phi) \equiv \mathrm{CheckExistsAlways}(\mathrm{Sat}(\phi))$

We compute $\mathrm{Sat}(\exists(\phi_1 \cup \phi_2))$. Start from $T := \mathrm{Sat}(\phi_2)$, compute the least fixed point of $F(T) = T \cup \{s \in \mathrm{Sat}(\phi_1) \mid \mathrm{Post}(s) \cap T \neq \emptyset\}$, effectively adding predecessors of states in T that satisfy ϕ_1. This is based on the idea that $\exists(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists(\phi_1 \cup \phi_2))$.

In a similar fashion, we can compute $\mathrm{Sat}(\exists\Box\phi)$ given $\mathrm{Sat}(\phi)$ by taking the expansion law

$$\exists\Box\phi \equiv \phi \wedge \exists \bigcirc \exists\Box\phi$$

Then starting with $T := \text{Sat}(\phi)$, we can take the greatest fixed point of the function $F(T) = T \cap \{s \in \text{Sat}(\phi) \mid \text{Post}(s) \cap T \neq \emptyset\}$. Alternatively, model checking on $\exists\Box\phi$ is essetially elements along paths that end up on strongly connected components of the subgraph induced by ϕ.

Given a LTS M and CTL formula ϕ, determining $M \models \phi$ is in $O(|M| \cdot |\phi|)$. where $|M|$ is the number of states + number of transitions (size of graph) and $|\phi|$ is the number of operators in ϕ. The worst case is given when all operators are temporal operators where each requires a single traversal of the whole model.

5.1.1 Symbolic Model Checking via BDDs

Definition 5.1.5. *Suppose we have an encoding of X into n Boolean variables (in particular, this is always possible for a finite set X). Consider a state space X and a subset $X' \subseteq X$. We define a* ***characteristic function*** $1_{X'} : X \to \{0,1\}$ *to be the function that maps to 1 if and only if $x \in X'$. Then, $1_{X'}$ induces a function $f_{X'}(x_1, \ldots, x_n) : \{0,1\}^n \to \{0,1\}$. Then, $f_{X'}$ can be represented by a corresponding BDD, which we write $B_{X'}$.*

Example 5.1.6. Given an encoding over a state space S and a subset $S' \subseteq S$, we have an induced function $f_{S'} : \{0,1\}^n \to \{0,1\}$ and a BDD $B_{S'}$. Similarly, a relation $\to \subseteq S \times S$ has an induced function $f_\to : \{0,1\}^{2n} \to \{0,1\}$ and a corresponding BDD $B_\to$.

For efficiency reasons, the variable ordering over a relation where $x = x_1, \ldots, x_n$ and $x' = x'_1, \ldots, x'_n$ is given by the function $f_\to(x_1, x'_1, \ldots, x_n, x'_n) : \{0,1\}^{2n} \to \{0,1\}$.

We can also represent this specific kind of BDD by a $2^n \times 2^n$ matrix, such that given a BDD M, we represent

$$M = \begin{pmatrix} M|_{x_i=0,x'_i=0} & M|_{x_i=0,x'_i=1} \\ M|_{x_i=1,x'_i=0} & M|_{x_i=1,x'_i=1} \end{pmatrix}$$

such that at the end of the sub-matrix process, we have the corresponding value obtained by $f_\to$.

Repeated submatricies of the above form are represented by a shared BDD node (as they are isomorphic). Simple matricies like I_{2^n} are represented by small BDDs (size $3n+1$), and the constant matrix is represented by 1 element.

Notation 5.1.7. We write $\exists x.B$ to mean $\exists x.f_B$ which is defined to be $f_B[0/x] \vee f_B[1/x]$, where we equivalently write $B|_{x=0}$ and $B|_{x=1}$. Conjunction and disjunction are equivalent to taking minimum and maxmium.

Example 5.1.8. Consider

$$\begin{aligned} \text{Sat}(\exists \bigcirc \phi) &= \{s \in S \mid \text{Post}(s) \cap \text{Sat}(\phi) \neq \emptyset\} \\ &= \text{Pre}(\text{Sat}(\phi)) \end{aligned}$$

Now, $\text{Pre}(T) = \{s \in S \mid \exists s', s \to s' \wedge s' \in T\}$. Thus,

$$1_{\text{Pre}(T)}(s) = \exists s'(1_\to(s, s') \wedge 1_T(s'))$$

The induced map is

$$f_{\text{Pre}(T)} = \exists_{x'}(f_\to \wedge f_T[x'/x])$$

Giving

$$f_{\text{Sat}(\exists\bigcirc\phi)} = \exists_{x'}(f_\to \wedge f_{\text{Sat}(\phi)}[x'/x])$$

Example 5.1.9. $\mathrm{Sat}(\exists(\phi_1 \cup \phi_2))$ is the least fixpoint of the function

$$\begin{aligned} F(T) &= \mathrm{Sat}(\phi_2) \cup \{s \in \mathrm{Sat}(\phi_1) \mid \mathrm{Post}(s) \cap T \neq \emptyset\} \\ &= \mathrm{Sat}(\phi_2) \cup (\mathrm{Sat}(\phi_1) \cap \mathrm{Pre}(T)) \end{aligned}$$

The induced map is

$$f_{F(T)} = f_{\mathrm{Sat}(\phi_2)} \vee f_{\mathrm{Sat}(\phi_1)} \wedge \exists_{x'}(f_{\to} \wedge f_T[x'/x])$$

This induces a function between BDDs via

$$F(B) = B_{\mathrm{Sat}(\phi_2)} \vee (B_{\mathrm{Sat}(\phi_1)} \wedge \mathrm{Pre}(B))$$

which we can compute the least fixed point of.

Example 5.1.10. Suppose we have programs

```
process Flip₁ = while true do (if x₁ = x₂ then x₁ := 1 - x₁) od
process Flip₂ = while true do x₂ := 1 - x₂ od
```

And $M = [[\mathrm{Flip}_1 \mid\mid\mid \mathrm{Flip}_2]] = [[\mathrm{Flip}_1] \mid\mid\mid [\mathrm{Flip}_2]]$. As an LTS, this is

$$\begin{tikzcd} (0,0) \arrow[r, shift left] \arrow[d] & (0,1) \arrow[l, shift left] \\ (1,0) \arrow[r, shift left] & (1,1) \arrow[l, shift left] \arrow[u] \end{tikzcd}$$

Then, writing $B_{\mathrm{Flip}_1} = (x_1 \leftrightarrow x_2) \wedge (x_1' \leftrightarrow \neg x_1)$ and $B_{\mathrm{Flip}_2} = x_2' \leftrightarrow \neg x_2$, we have that

$$B_M = (B_{\mathrm{Flip}_1} \wedge B_{\mathrm{Id}_2}) \vee (B_{\mathrm{Flip}_2} \wedge B_{\mathrm{Id}_1})$$

Remark 5.1.11.

$$\begin{tikzcd} \text{Program} \arrow[dr] & & \text{Matrix } \mathcal{M}_M \arrow[dr] & \\ & \text{LTS } M \arrow[ur] \arrow[rr] & & \text{BDD } B_M \\ \text{CTL Sat}(\phi) \arrow[urrr] & & & \end{tikzcd}$$

We can transform programs and properties about programs into a BDD (which practically, we hope are compact). The nice property is that then we can manipulate programs and sets at a time over BDDs, and membership testing is simply a run through the BDD.

5.1.2 Bounded Model Checking

The basic idea is to unroll a model up to a fixed depth k, searching for counterexamples until this depth, reducing model checking to SAT. This method is sound but incomplete on its own (as the search size is limited).

Example 5.1.12. Consider the CTL formula $\forall\Box a$. A finite path $s_0, \ldots, s_k$ such that s_k satisfies $\neg a$ is a counterexample.

Consider the 2-bit counter:

$$\begin{tikzcd} {} \arrow[d] & \\ (00) \arrow[d] & (11) \arrow[l] \\ (01) \arrow[r] & (10) \arrow[u] \end{tikzcd}$$

We can encode states via two boolean variables $x = x_0, x_1$ corresponding to the values of the first and second character on the state. Given two copies of variables x, x', we have the translation relation

$$T(x_0, x_1, x_0', x_1') = (\neg x_0 \wedge \neg x_1 \wedge \neg x_0' \neg x_1') \vee \cdots$$

where the first term corresponds to the transition from 00 to 01. Equivalently,

$$T(x_0, x_1, x_0', x_1') = (x_0' \leftrightarrow \neg(x_0 \leftrightarrow x_1)) \wedge (x_1' \leftrightarrow \neg x_1)$$

Let a be the statement that 'the counter is less than 3'. Suppose now that we fix $k = 1$. The initial state is $\mathrm{init}(x_0, x_1) = \neg x_0 \wedge \neg x_1$. The invariant is $\mathrm{inv}(x_0, x_1) = \neg(x_0 \wedge x_1)$ and $\mathrm{inv}'(x_0', x_1') = \neg(x_0', x_1')$. Then,

$$\phi_1(x_0, x_1, x_0', x_1') = (\mathrm{init} \wedge \neg\mathrm{inv}) \vee (\mathrm{init} \wedge T \wedge \neg\mathrm{inv}')$$

is satisfiable if there exists a counterexample of length less at most 1. Extending this in the natural way gives a way to write ϕ_k which is satisfied if and only if there is a counterexample of length at most k.

Example 5.1.13. A counterexample for $\forall\Diamond a$ is a finite path $s_0, \ldots, s_m, \ldots, s_n$ for some $m \geq 0$ and $n > m$ such that s_i all satisfy $\neg a$ and $s_m = s_n$.

Using notation from the previous example, assigning corresponding invariants,

$$\phi_1 = \mathrm{init} \wedge T_0 \wedge (x_1 = x_0) \wedge \neg\mathrm{inv}_0$$

where x_0, x_1 represent states, and equality is a conjunction of $\leftrightarrow$ on each binary variable. Generally,

$$\phi_k = \mathrm{init} \wedge (T_0 \wedge \cdots \wedge T_{k-1}) \wedge \bigvee_{0 \leq h < k} (x_k = x_h) \wedge (\neg\mathrm{inv}_0 \wedge \cdots \wedge \neg\mathrm{inv}_{k-1})$$

which represents a path of length k that fails to satisfy the invariant.

Definition 5.1.14. *A **completeness threshold** is an integer k for a CTL ϕ and LTS M such that if there is no counterexample of length at most k, then $M \models \phi$, which we write as $M \models_k \phi$.*

Given a state s of M, we write $s \models_k \phi$ if there does not eixst a counterexample for ϕ of length at most k starting at s.

In particular, k is a completeness threshold for M, ϕ, if $M \models_k \phi$ implies $M \models \phi$. For instance, the completeness threshold for $\bigcirc a$ is 1.

Remark 5.1.15. $\models_k$ cannot be defined inductively as for normal CTL semantics (Proof??). $s \models_k \phi$ is in general not equivalent to $s \models_k \neg\phi$. For instance, $s \models_0 \exists \bigcirc a$ and $s \models_0 \neg\exists \bigcirc a$, as a path of length 0 is not able to faisify claims that are completely about the future.

Proposition 5.1.16. *Given a CTL formula $\forall\Box a$, $d(M)$ is a completeness threshold. Similarly, for a formula $\forall\Diamond a$, $rd(M)$ is a completeness threshold.*

Proof. If there is a counterexample for $\forall\Box a$, it must be due to $\neg a$ on a reachable state from I. This state is reachable with a path length at most $d(M)$.

If there is a counterexample for $\forall\Diamond a$, we should find a finite path $s_0, \ldots, s_n$ that is always $\neg a$ and $s_n \to s_i$ for $0 \le i \le n$. Given such counterexample, we can always shorten it to one which has no duplicating states (is loop free). This is a counterexample of length at most $rd(M)$.

The path of length $rd(M)$ can visit □

Generally we have,

- $\mathrm{CT}(M, \text{true}) = \mathrm{CT}(M, \text{false}) = \mathrm{CT}(M, a) = \mathrm{CT}(M, \neg a) = 0$
- $\mathrm{CT}(M, \phi_1 \wedge \phi_2) = \mathrm{CT}(M, \phi_1 \vee \phi_2) = \max(\mathrm{CT}(M, \phi_1), \mathrm{CT}(M, \phi_2))$
- $\mathrm{CT}(M, \exists \bigcirc \phi) = \mathrm{CT}(M, \forall \bigcirc \phi) = 1 + \mathrm{CT}(M, \phi)$
- $\mathrm{CT}(M, \exists(\phi_1 \cup \phi_2)) = \mathrm{CT}(M, \forall(\phi_1 \cup \phi_2)) = rd(M) + \mathrm{CT}(M, \phi_2)$

where $\mathrm{CT}(M, \phi)$ is the (canonical) completeness threshold for M and ϕ. So one can always convert a CTL to PNF and find the completeness threshold by computing the recurrence diameter.

The final one uses the fact that if there exists a path in which it is not the case that $\phi_1 \cup \phi_2$, then we have (TODO!!!!!)

5.2 LTL

Definition 5.2.1. *A **counterexample** for a LTL formula ψ on a LTS M is a sufficiently long prefix of a path π of M with $\pi \not\models \psi$ which is sufficiently long to show $\pi \not\models \psi$.*

Example 5.2.2. Some examples of counterexamples on LTL:

- A counterexample for $\Box a$ is a finite path ending in $\neg a$
- A counterexample for $\bigcirc a$ is a 2-state path ending in $\neg a$
- A counterexample for $\Diamond a$ is a finite prefix of $\neg a$ states followed by a single cylce of $\neg a$ states
- A counterexample of $A \wedge B$ is a counterexample for A or B
- A counterexample for $\Box a \to \Box b$ is a finite prefix of a states followed by a single cycle of a states (to show $\Box a$) with one state where b is false (to show $\neg\Box b$).

Remark 5.2.3. If a linear-time property can be written as a regular language over 2^{AP}, then there is a NFA representation over 2^{AP}.

Definition 5.2.4. *A **regular safety property** is a $P_{\mathrm{safe}} \subseteq (2^{\mathrm{AP}})^\omega$ is a regular safety property if the language*

$$\{w \in (2^{\mathrm{AP}})^* \mid \forall\sigma \in (2^{\mathrm{AP}})^\omega, w\sigma \notin P_{\mathrm{safe}}\}$$

is regular.

Example 5.2.5. Consider the NFA $\mathcal{A}$,

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[loop above] \arrow[r] & q_1 \end{tikzcd}$$

where q_1 is the accepting state, q_0 has label $\emptyset$, q_1 has label $\{\text{fail}\}$. Then the language by the NFA is $\mathcal{L}(\mathcal{A}) = \{\{\text{fail}\}, \emptyset\{\text{fail}\}, \emptyset\emptyset\{\text{fail}\}, \cdots\}$, which can translate to 'finite traces where a failure occurs'. This is the regular language that corresponds to the (minimal) bad prefixes of $\Box\neg\text{fail}$.

Example 5.2.6. The regular safety property 'at most 2 failures occur' has a regular expression for bad prefixes,

$$(\neg\text{fail})^*.\text{fail}.(\neg\text{fail})^*.\text{fail}.(\neg\text{fail})^*.\text{fail}$$

Remark 5.2.7. Given an LTS M and a regular safety property P_{safe},

$$\begin{aligned} M \models P_{\text{safe}} &\iff \text{Traces}(M) \subseteq P_{\text{safe}} \\ &\iff \text{Traces}_{\text{fin}}(M) \cap \text{BadPref}(P_{\text{safe}}) = \emptyset \end{aligned}$$

Given an NFA $\mathcal{A}$ representing the bad prefixes, we have

$$M \models P_{\text{safe}} \iff \text{Traces}_{\text{fin}}(M) \cap \mathcal{L}(\mathcal{A}) = \emptyset$$

Definition 5.2.8. *Given an LTS $M = (S, \text{Act}, \to, I, \text{AP}, L)$ and an NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, we can construct the product LTS of M and $\mathcal{A}$, denoted $M \otimes \mathcal{A}$ as the LTS $(S \times Q, \text{Act}, \to', I', \{accept\}, L')$, where*

- *$I' = \{(s_0, q) \mid s_0 \in I$ and $q_0 - L(s_0) \to q$ for some $q_0 \in Q_0\}$*
- *$L'((s, q)) = \{accept\}$ if $q \in F$ and empty otherwise*
- $$\frac{s - \alpha \to' s' \wedge q - L(s') \to' q'}{(s, q) - \alpha \to' (s', q')}$$

The intuition is that states correspond to (states in $M \to$ states in $\mathcal{A}$), where a transition $s \to s'$ on the LTS induces a transition on the NFA along the label $L(s')$ (hence the requirement in the initial states). The acceptance condition is that the induced path by the LTS reaches an accepting state on $\mathcal{A}$.

Consequently, we have

$$\begin{aligned} M \models P_{\text{safe}} &\iff \text{Traces}_{\text{fin}}(M) \cap \mathcal{L}(\mathcal{A}) = \emptyset \\ &\iff M \otimes \mathcal{A} \models \Box\neg\text{accept} \end{aligned}$$

So model checking becomes a problem of reachability on graphs. Thus $M \not\models P_{\text{safe}}$ if and only if some path satisfies $\Diamond\text{accept}$ in $M \otimes \mathcal{A}$.

Definition 5.2.9. *$P \subseteq (2^{\text{AP}})^\omega$ is an ω-regular property if P is an ω-regular language over 2^{AP}.*

Lemma 5.2.10. *Any regular safety property is an ω-regular property.*

Lemma 5.2.11. *Any LTL formula corresponds to an ω-regular property.*

Definition 5.2.12. *Given an LTS M and NBA $\mathcal{A}$, the product denoted $M \otimes \mathcal{A}$ is the same object as the construction with NFAs.*

Remark 5.2.13. We have the relation,

$$\begin{aligned} M \models \psi &\iff \text{Traces}(M) \subseteq \text{Words}(\psi) \\ &\iff \text{Traces}(M) \cap \text{Words}(\neg\psi) = \emptyset \\ &\iff \text{Traces}(M) \cap \mathcal{L}_\omega(\mathcal{A}_{\neg\psi}) = \emptyset \\ &\iff \text{there is no accepting path (cycle) in } M \otimes \mathcal{A}_{\neg\psi} \\ &\iff M \otimes \mathcal{A} \models \Diamond\Box\neg\text{accept} \end{aligned}$$

That is, to model check an LTL formula ψ on an LTS M, it is sufficient to check in the LTS-NBA-product. Also, we can see that $M \not\models \psi$ if and only if there is some path that satisfies $\Box\Diamond$accept in $M \otimes \mathcal{A}_{\neg\psi}$.

Also note that this has structural similarity to checking regular safety properties on an LTS with a NFA. Regular safety properties are definitionally a subclass of ω-regular properties, and many can be represented as an LTL. In that case, the same model checking works there.

There are a few methods to check this, like search for non-trivial SCCs containing the accepting state, finding all accept states and DFS to find back edges, etc.

For a general LTL formula, we can convert to a GNBA, then to a corresponding NBA.

Lemma 5.2.14. *For any LTL ψ, there is a corresponding GNBA $\mathcal{A}$.*

Proof. Sketch. The states are sets of ψ's subformulae and their negations. $\square$

Remark 5.2.15. The time complexity for LTL model checking is $O(|M| \cdot 2^{|\psi|})$. Note that there are LTL formulas ψ whose NBA $A_{\neg\psi}$ is of size $O(2^{|\psi|})$, and checking for cycles can be done in linear time.

5.3 Software Model Checking

Definition 5.3.1. *A **single static assignment** (SSA) form is an intermediate representation of programs such that every assignment of a variable uses a new xi, with the access of a variable using the latest version. For convinience, we assume an initial value $x0$ for each variable.*

Example 5.3.2. The program

```
    x := 2 * y;
    x := x * z;
    x++;
```

converts to

```
    x1 := 2 * y0;
    x2 := x1 * z0;
    x3 := x2 + 1;
```

If we have conditionals, we write separate SSA code for each branch (with fresh names), and resolve the branch afterwards. Assertions are predicated by guards.

Example 5.3.3. A code like

```
x := y;
if (x > z) {
    x := x + 1;
    assert x > y;
    y = y + 1;
} else {
    x := x + y;
    assert x ≠ y;
}
```

converts to

```
x1 := y0;
x2 := x1 + 1;
assert(x1 > z0) ⇒ (x2 > y0);
y1 := y0 + 1;
x3 := x1 + y0;
assert !(x1 > z0) ⇒ (x3 ≠ y0);
x4 := (x1 > z0) ? x2 : x3;
y2 := (x1 > z0) ? y1 : y0;
```

From here, we convert to a CNF ϕ such that it can be satisfied if and only if a program execution can violate an assertion. Specifically, one conjunctively equates every line (T) and conjuct that with the negation of the assertion (invariant). Thus, $\phi = T \land \neg \text{inv}$. Note that these equations are not over boolean variables but predicates (of equality).

To check for satisfiability of formulae over predicates, we can use bit blasting (which converts integers to binary, variables from predicates to boolean variables) and solve using the standard SAT solver. Alternatively, we can use a satisfiability modulo theory (SMT) solver.

To convert from a program with loops, first simplify the program such that for loops are converted to while loops, and structure becomes simpler (like changing breaks to gotos). Then, we unwind loops to a fixed depth k. For instance, we can convert

```
while (condition) {
    body
}
statements
```

into

```
if (condition) {
    body
    if (condition) {
        body
        assume (!condition);
    }
}
statements
```

where the above is a case of 2 unwindings and replacing the while loop by a condition that blocks an execution if conditions. Consequently, this is sound but not complete. To check for completeness, we assert in the body that (!condition), as UNSAT implies there are no bugs. If the assertion is violated, we may increase k.

The key is to mimic program execution symbolically, using symbols for unknown variables.

Definition 5.3.4. *A **execution tree** or a **computation tree** is a binary tree where internal nodes are branches (like if conditions) and leaf nodes are program exits, extracted from the control flow graph.*

The execution tree is typically infinite in size, but one can partially explore the execution tree. Each control flow path (ending in the assertion) is executed symbolically, constructing the path condition, appending the negation of the assertion to be checked. If the path condition is satisfiable, then there is a assertion violation.

For instance, the code

```
void f(int x) {
    int y = 0;
    if (x >= 1000) {
        y++;
        x--;
    }
    if (x < 1000) {
        y--;
    }
    assert(y != 0);
}
```

has 4 control paths, corresponding to branches in each if condition. Of course we can satisfy the negation of the assertion when both conditions are true.

Program with loops have infinitely many paths, so there needs to be good heuristics to explore paths.

6 Equivalence

6.1 Bisimulation

Definition 6.1.1. *Let $M_i = (S_i, \to_i, I_i, \mathrm{AP}, L_i)$ be two LTS over the same atomic propositions for $i = 1, 2$. Then, a **bisimulation** (between LTSs) is a binary relation $R \subseteq S_1 \times S_2$ such that*

- *$\forall s_1 \in I_1, \exists s_2 \in I_2$ such that $(s_1, s_2) \in R$ and vice versa.*
- *For all $(s_1, s_2) \in R$, $L_1(s_1) = L_2(s_2)$ and if $s_1' \in \mathrm{Post}(s_1)$, then there exists a $s_2' \in \mathrm{Post}(s_2)$ such that $(s_1', s_2') \in R$ and vice versa.*

*If such a relation exists, we write $M_1 \sim M_2$ and that M_1 and M_2 are **bisimular.***

Definition 6.1.2. *Let $M = (S, \to, I, \mathrm{AP}, L)$ be an LTS. A **bisimulation** (between states) over M is a binary relation $R \subseteq S \times S$ such that for all $(s_1, s_2) \in R$,*

- *$L_1(s_1) = L_2(s_2)$*
- *if $s_1' \in \mathrm{Post}(s_1)$, then there exists $s_2' \in \mathrm{Post}(s_2)$ such that $(s_1', s_2') \in R$ and vice versa.*

If such an R exists and $(s_1, s_2) \in R$, we write $s_1 \sim_M s_2$, and we say that s_1 and s_2 are bisimular.

Definition 6.1.3. *Given two bisimulation relations R_1 and R_2, we say that R_1 is **coarser** than R_2 (and that R_2 is **finer** than R_1) if $s\ R_2\ t \implies s\ R_1\ t$ for all s, t. Alternatively, we have that $R_2 \subseteq R_1$.*

Proposition 6.1.4. *The union of two bisimulations R_1 and R_2 is a coarser bisimulation relation.*

Proof. OoSN. □

Definition 6.1.5. *Given a bisimulation R, the induced quotient space $S/R = \{[s]_R \mid s \in S\}$, where $[s]_R = \{s' \in S \mid (s, s') \in R\}$. Given M and R, the **bisimulation quotient** (system) is*

$$M/R = (S/R, \to', I', \mathrm{AP}, L')$$

such that

- $I' = \{[s]_R \mid s \in I\}$
- $L'([s]_R) = L(s)$
- $\dfrac{s \to s'}{[s]_R \to' [s']_R}$

Lemma 6.1.6. *For any M and bisimulation R, M/R is bisimular to M.*

Proof. OoSN. □

Lemma 6.1.7. *Let $\sim_M$ be the union of all bisimulations over M. Then $M/\sim_M$ is the obtained as the coarsest among all possible bisimulations R over M.*

Proof. OoSN. □

We sometimes call this the bisimulation quotient for M.

Lemma 6.1.8. *If $M_1 \sim M_2$, then* $\mathrm{Traces}(M_1) = \mathrm{Traces}(M_2)$.

Proof. The idea is that for any path in M_1 has a corresponding path in M_2 that behaves identically over any finite observation (by induction). We can sort-of extend this by a chain-completeness argument. □

Corollary 6.1.9. *Any bisimular LTSs satisfy the same linera-time properties.*

Theorem 6.1.10 (Equivalence Results).

- $\equiv_{\text{trace}} \subsetneq \equiv_{\text{LTL}}$ *(strictness requires infinite state space)*
- $\sim_M = \equiv_{\text{CTL}^*}$*; bisimulation equivalence and CTL* equivalence coincides.*
- $\equiv_{\text{CTL}} = \equiv_{\text{CTL}^*}$.

Proof. OoSN. □

Remark 6.1.11. As a consequence of the second result, bisimulation preserves satisfaction of CTL*, CTL, and LTL formulae. In particular, it suffices to model check on the quotient model, for both $M \models \phi$ and $M \not\models \phi$. Also, given any $s \sim_M s'$, $s \models \phi$ and $s \not\models \phi$ by any CTL formula disproves bisimilarity.

Example 6.1.12. Consider two LTSs,

$$\begin{tikzcd}
& {} \arrow[d] & & & {} \arrow[d] & \\
& s_0 \arrow[d] & & & s_0 \arrow[dl] \arrow[dr] & \\
& s_1 \arrow[dl] \arrow[dr] & & s_1 \arrow[d] & & s_1 \arrow[d] \\
s_2 \arrow[uur] & & s_3 \arrow[uul] & s_2 \arrow[uur] & & s_3 \arrow[uul]
\end{tikzcd}$$

which we have given labels directly on the state. Then the CTL formula

$$\phi = \exists \bigcirc (\exists \bigcirc s_2 \land \exists \bigcirc s_3)$$

is satisfied by the left but not by the right.

Remark 6.1.13. Checking whether R is a bisimulation relation is straightforward. Equivalence checking (seeing whether M_1 and M_2 are bisimular) can be reduced to a problem of finding the bisimulation quotient of the disjoint union of LTSs M_1 and M_2, and ensuring that for any $s_1 \in I_1$ we can find an $s_2 \in I_2$ such that $s_1 \sim s_2$ and vice versa.

One algorithm to compute the bisimulation quotient of an LTS is as follows:

- start with the partition based on labelling
- repeatedly split state blocks that are not bisimilar

at termination, this gives the coarsest partition.

Example 6.1.14. Suppose there is some simple process who is composed in parallel 100 times. The state space becomes exponential. However, without collapsing from this LTS, we can instead count the number of processes in each local state via counting, choosing a representative via sorting (by giving a order on local states). This gives a bisimular LTS that is significantly smaller, and is easier to find the bisimulation quotient.

6.2 Simulation

Definition 6.2.1. *Let $M_i = (S_i, \rightarrow_i, I_i, \text{AP}, L_i)$ be two LTS over the same atomic propositions for $i = 1, 2$. Then, a **simulation** (between LTSs) is a binary relation $R \subseteq S_1 \times S_2$ such that*

- $\forall s_1 \in I_1, \exists s_2 \in I_2$ *such that* $(s_1, s_2) \in R$.
- *For all $(s_1, s_2) \in R$, $L_1(s_1) = L_2(s_2)$ and if $s_1' \in \text{Post}(s_1)$, then there exists a $s_2' \in \text{Post}(s_2)$ such that $(s_1', s_2') \in R$.*

*If such a relation exists, we write $M_1 \preceq M_2$ and that M_1 is **simulated** by M_2.*

Example 6.2.2. Taking the LTSs from Example 6.1.12, we clearly have that the right side $\preceq$ left side.

Definition 6.2.3. *Let $M = (S, \rightarrow, I, \text{AP}, L)$ be an LTS. A **simulation** (between states) over M is a binary relation $R \subseteq S \times S$ such that for all $(s_1, s_2) \in R$,*

- $L_1(s_1) = L_2(s_2)$
- *if* $s_1' \in \text{Post}(s_1)$*, then there exists* $s_2' \in \text{Post}(s_2)$ *such that* $(s_1', s_2') \in R$*.*

If such an R *exists and* $(s_1, s_2) \in R$*, we write* $s_1 \preceq_M s_2$*. This is naturally a preorder.*

Definition 6.2.4. *Let* M *be an LTS and* A *be a set of abstract states. Let* $f : S \to A$ *be an* ***abstraction function*** *such that for all* $s, s' \in S$*,* $f(s) = f(s')$ *implies* $L(s) = L(s')$*. Then, define the* ***abstract LTS****, written* M_f *to be*

$$M_f = (A, \to_f, I_f, \text{AP}, L_f)$$

such that

- $I_f = \{f(s) \mid s \in I\}$
- $L_f(f(s)) = L(s)$
- $\dfrac{s \to s'}{f(s) \to_f f(s')}$

Proposition 6.2.5. *Let* M_f *be an abstract LTS of* M*. Then we have* $M \preceq M_f$*.*

Remark 6.2.6. A could be a partition of S, but is not required. Note that there is a trivial f that partitions based on the label.

Definition 6.2.7. *Given abstraction functions* f *and* f^r*,* f^r *refines* f *if*

- *for all* $s, t \in S$*,* $f^r(s) = f^r(t)$ *implies that* $f(s) = f(t)$
- *there exists* $s, t \in S$ *such that* $f^r(s) \neq f^r(t)$ *and* $f(s) = f(t)$

Definition 6.2.8. *If LTS* M_1, M_2 *satisfy* $M_1 \preceq M_2$ *and* $M_2 \preceq M_1$*, we say that they are a* ***simulation equivalence*** *and that* $M_1 \simeq M_2$*.*

Example 6.2.9. Simulation equivalence need not imply bisimularity. Consider the LTSs,

$$\begin{tikzcd}
{} \arrow[d] & & & {} \arrow[d] \\
s_0 \arrow[d] & & & s_0 \arrow[dl] \arrow[d] \\
s_1 \arrow[d] \arrow[loop right] & & s_1 \arrow[dr] & s_1 \arrow[d] \arrow[loop right] \\
s_2 \arrow[loop right] & & & s_2 \arrow[loop right]
\end{tikzcd}$$

where we show labels instead of states. This is clearly simulation equivalent but not bisimular.

Lemma 6.2.10. *Simulation implies trace inclusion. That is,*

$$M_1 \preceq M_2 \implies \text{Traces}(M_1) \subseteq \text{Traces}(M_2)$$

In particular, if $M_1 \preceq M_2$*, for any LT property* P*,* $M_2 \models P$ *implies* $M_1 \models P$*.*

Definition 6.2.11. *Let* $\forall\text{CTL}^*$ *be the positive normal fragments of CTL* without* $\exists$.

Lemma 6.2.12. *Every safety property in CTL* is in* $\forall CTL^*$.

Lemma 6.2.13. *For any* $\forall\text{CTL}^*$ *formula* ϕ,

- $M_1 \preceq M_2$ *if and only if* $M_2 \models \phi \implies M_1 \models \phi$
- $M_1 \simeq M_2$ *if and only if* $M_2 \models \phi \iff M_1 \models \phi$

Corollary 6.2.14. *Given LTS* M_1, M_2 *such that* $M_1 \preceq M_2$ *and a* $\forall\text{CTL}^*$ *formula such that* $M_1 \models \phi$ *and* $M_2 \not\models \phi$*, then* $M_1 \not\simeq M_2$.

Remark 6.2.15. Considering $\forall\text{CTL}^*$ (positive normal fragments of CTL* without $\exists$), we have

The CEGAR (counterexample-guided abstraction refinement) algorithm does the following, given an LTS M and property ϕ (in LTL or $\forall[CTL]^*$):

- Start with an abstraction M'
- Check $M' \models \phi$. If yes, return $M \models \phi$
- Generate counterexample c and try to map to M. If c holds in M (symbolically mappable), return $M \not\models \phi$
- Else, counterexample is 'spurious,' (or infeasible) so refine M' based on the counterexample and repeat.

The idea is that we start with an overapproximation and move towards smaller overapproximations by finding counterexamples in the overapproximation. We can implement this in a SAT-based manner, which uses sat to model check $M' \models \phi$ and k-step SAT the counterexamples.

Example 6.2.16. Consider the LTS

$$M_1 = \begin{tikzcd} {} \arrow[d] & \\ s_1 \arrow[d] & s_2 \arrow[loop right] \\ s_1 \arrow[ur] \arrow[r] & s_2 \arrow[loop right] \end{tikzcd} \qquad M_2 = \begin{tikzcd} {} \arrow[r] & s_1 \arrow[loop above] \arrow[r] & s_2 \arrow[loop above] \end{tikzcd}$$

and the initial abstraction M_2 such that $M_1 \preceq M_2$. Suppose we want to test $M_1 \models \forall\Diamond s_2$. We have a counterexample in M_2 (consider t_1^ω), but this is spurious, so refine M_2

$$M_3 = \begin{tikzcd} {} \arrow[r] & s_1 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[loop above] \end{tikzcd}$$

such that $M_1 \preceq M_3$. Now $M_3 \models \forall\Diamond s_2$ so $M_1 \models \forall\Diamond s_2$.

Example 6.2.17. Consider

$$M_1 = \begin{tikzcd} {} \arrow[d] & \\ s_1 \arrow[d, shift left] & s_2 \arrow[loop right] \\ s_1 \arrow[u, shift left] \arrow[ur] \arrow[r] & s_2 \arrow[loop right] \end{tikzcd} \qquad M_2 = \begin{tikzcd} {} \arrow[r] & s_1 \arrow[r] \arrow[loop above] & s_2 \arrow[loop above] \end{tikzcd}$$

Then we have $M_2 \not\models \forall\Box\neg s_2$ with counterexamples like $s_1 s_2$ (slight abuse of notation, assuming we know the original labels) or $s_1 s_1 s_2$. The first is spurious but the latter is not.