Automated Reasoning and Formal Verification

Diego Oniarti

Anno 2024-2025

Contents

1 25-02-2025

intro

Slides will be on his webpage along with the recordings.

The exam will consist of a script and an oral exam on the topics of the whole course.

boolean/propositional logic

A propositional **formula** can be:

- $\top$, $\bot$
- Propositional **atoms** $A_1, A_2, \ldots, A_n$
- A combination of other formulas. If φ_1 and ϕ_2 are formulas, so are:
  - $\neg\varphi_1$
  - $\varphi_1 \wedge \phi_2$
  - $\varphi_1 \vee \phi_2$
  - $\varphi_1 \rightarrow \phi_2$
  - $\varphi_1 \leftarrow \phi_2$
  - $\varphi_1 \leftrightarrow \phi_2$
  - $\varphi_1 \oplus \phi_2$

We define a function $Atoms(\varphi)$ representing the set $\{A_1, \ldots, A_n\}$ of atoms in ϕ

A **clause** is a disjunction of literals $\bigvee_j l_j$ or $(A_1 \vee \neg A_2 \vee ...)$

A **cube** is a conjunction of literals $\bigwedge_j l_j$ or $(A_1 \wedge \neg A_2 \wedge ...)$

trees and DAGS

A tree is a natural representation of an expression, but in the worst cases it can grow exponentially. The same information about the formula can be conveyed by a *Directed Acyclic Graph*, which can grow linearly in size.

Total Truth Assignment

They can also be abbreviated as *Total Assignment*.
A total truth assignment $\mu : Atoms(\varphi) \mapsto \{\top, \bot\}$ represents *one* possible state of the formula.

Partial Truth Assignment

A partial truth assignment $\mu : \mathcal{A} \mapsto \{\top, \bot\}, \mathcal{A} \subset Atoms(\varphi)$ represents 2^k total assignments, where k is the number of unassigned literals.

μ defined for total and partial truth assignments a can be seen as a set of literals (positive and negative ones) or a formula.

Set of models

$M(\varphi) \triangleq \{\mu | \mu \models \phi\}$ is the set of all models of ϕ.

Properties

- φ is *valid* if every μ models ϕ
- φ valid $\iff$ $\neg\phi$ unsatisfiable
- $\alpha \models \beta \iff \alpha \rightarrow \beta$ valid Deduction theorem
- corollary $\alpha \models \beta \iff \alpha \wedge \neg\beta$ not satisfiable

Equivalence and Equi-satisfiability

α and β are *equivalent* if $\forall\mu.\mu \models \alpha \iff \mu \models \beta$.
In other terms, $M(\alpha) = M(\beta)$.

Equi-satisfiability $M(\alpha) \neq \emptyset \iff M(\beta) \neq \emptyset$. This property is mostly used when applying transformations to formulas $\beta \triangleq T(\alpha)$.

Transformations can be *validity preserving* if they preserve the validity of the formula they're being applied to, or *satisfiability preserving* if they preserve its satisfiability.

Shannon's expansion

$$\exists v.\varphi := \phi|v = \bot \vee \phi|v = \top$$

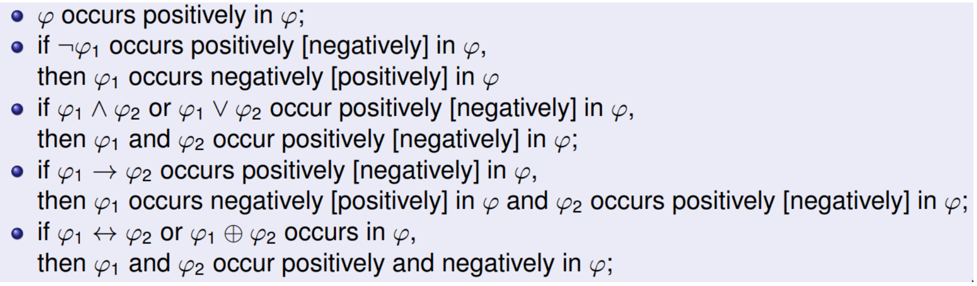
The existential is a disjunction between two possible formulas. One where v is set to true, and one where it is set to false.

$$\forall v.\varphi := \phi|v = \bot \wedge \phi|v = \top$$

The universal one is similar, with a conjunction between the two.

Polarity of subformulas

Polarity is a metric defined for each subformula of a formula φ that tells us under how many nested negations it occurs. It can either be positive, negative, or both in some cases.
The recursive rules to determine the polarity are shown in the image below



- φ occurs positively in φ;
- if $\neg\varphi_1$ occurs positively [negatively] in φ, then φ_1 occurs negatively [positively] in φ
- if $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ occur positively [negatively] in φ, then φ_1 and φ_2 occur positively [negatively] in φ;
- if $\varphi_1 \rightarrow \varphi_2$ occurs positively [negatively] in φ, then φ_1 occurs negatively [positively] in φ and φ_2 occurs positively [negatively] in φ;
- if $\varphi_1 \leftrightarrow \varphi_2$ or $\varphi_1 \oplus \varphi_2$ occurs in φ, then φ_1 and φ_2 occur positively and negatively in φ;

If we assume $\top = 1, \bot = 0$ we can also see the polarity of a subformula as "how much it contributes to the overall value of the formula".

2 Normal forms

2.1 Negative Normal Form - NNF

A negative normal form is a formula in which each negations has been pushed down to the atoms. This implies that every subformula in $NNF(\varphi)$ has positive polarity.

Properties

- Every formula can be made into negative normal form
- NNF transformation preserves equivalence

2.2 Conjunctive Normal Form - CNF

$$\bigvee_{i=1}^{L} \bigwedge_{j=1}^{K_i} l_{ij}$$

$$(l_{11} \wedge l_{12}) \vee (l_{21} \wedge l_{22} \wedge l_{23}) \vee (...) \vee ...$$

Every formula can be converted in *Conjunctive Normal Form*, but there are different ways to do so.

2.2.1 Naive CNF conversion

The more intuitive and straightforward method consist of:

1. Expanding implications and equivalences
2. Pushing down negations like in NNF
3. Recursively applying DeMorgan's rule to get the CNF shape

This method produces a CNF that is equivalent to the original formula and has the same atoms. It is however rarely used in practical applications because it can be up to exponentially larger than the original formula.

2.2.2 Labeling CNF conversion

This is a more efficient *bottom-up* approach, which can be executed while parsing the expression.
The main idea is that of introducing new variables that serve as "*labels*" for each subformula. The smaller formulas can be converted to CNF with the naive approach, and then assembled through the labels.

This method introduces new atoms, but $\exists(B_1, \ldots, B_k).CNF_l(\varphi)$ equiv ϕ where $B_1, \ldots, B_k$ are the newly introduced variables. This means that ϕ and $CNF_l(\phi)$ are equisatisfiable.

The representation obtained from the CNF_l can be reduced further in size by using polarization to change some implications around.

3 Basic SAT-solving techniques

> **Example:** A classic problem is that of checking a query under a (usually much larger) knowledge base. This problem can be reduced to SAT. $KB \models \alpha$ or $M(KB) \subseteq M(\alpha)$
>
> $$KB \models \alpha \iff SAT(KB \vee \neg\alpha) = false$$

3.1 Intro - Unit propagation

Resolution rule Deduction of a new clause from a pair of clauses with *exactly* one incompatible variable (which is called the "*resolvent*").

$$(\underbrace{a}_{\text{common}} \vee \underbrace{b}_{\text{left}} \vee c) \wedge (\underbrace{a}_{\text{common}} \vee \underbrace{d}_{\text{right}} \vee \neg c) = (a \vee b \vee d)$$

We get $(common \vee left \vee right)$.

Removal of valid clauses If a clause is valid (always true) it can be removed from the formula.

Clause subsumption If a clause appears on its own and inside another clause, we can remove the second, bigger, clause.

$$(a \vee b) \wedge (c \vee a \vee b \vee d) = (a \vee b)$$

Unit resolution Having a clause composed of a single literal forces said literal to be true. This means we can remove all instances of the negated literal.

Unit subsumption Like clause subsumption but with a literal instead of a clause.

Unit propagation Is just the combination of unit resolution and unit subsumption.

These unit propagation rules can happen in a chain. After modifying the formula once we can create new unary clauses for example.

3.2 Resolution algorithm

Algorithm 1: Resolution algorithm

```
Assume input is in CNF;
$\varphi$ is a set of clauses;
//Search for a refutation of $\varphi$;
repeat
    apply resolution rule to pairs of clauses;
until a false clause is generated $\vee$ the rule is not applicable;
```

This algorithm is correct and complete, but operates in exponential memory and is time inefficient.

3.3 Tableaux

Search assignments satisfying φ by applying *elimination rules* on its connectors.
Try to be clever and put put smaller clauses first in the branching order.
The algorithm ends when we reach a leaf (SAT) or we get stuck in every branch (not SAT).
A branch is "stuck" when the path to reach it contains both l and $\neg l$.

Tableaux are handy because they only need the elimination rules to be defined and they can be done by hand. If the formula is in CNF we only really care about the $\vee$ and $\wedge$ elimination rules (respectively $\vee$ branches the formula and $\wedge$ stacks its parts on top of one another).

This is not efficient but it's still better than the resolution-based algorithm. It's also interesting seeing Tableaux as *semantic* resolution methods, since every step keeps memory of the previous ones.

3.4 DPLL - Davis-Putnam-Longeman-Loveland procedure

The DPLL procedure tries to build a truth assignment μ satisfying φ, and it does this by progressively assigning atoms.

> **Terminology.** A literal l is *pure* if it occurs only positively if it occurs only positively

The procedure relies on three rules:

$$\frac{\varphi \wedge (l)}{\varphi[l|\top]}\text{Unit} \quad \frac{\varphi}{\varphi[l|\top]}\text{l pure} \quad \frac{\varphi}{\varphi[l|\top] \quad \varphi[l|\bot]}\text{split}$$

After setting l to true, for example, remove all clauses containing l and all instances of $\neg l$.

For this technique it is important to find good heuristics when going to choose the next literal.

3.4.1 Backtracking problem

The main issue here is the "chronological" backtracking. A DPLL solver stores the assignments of literals on a stack, and once it reaches a dead branch it pops one element from this stack.
The issue with this method is that, if the true element to be changed is far into the stack, it will take a long time before reaching it (since it has to explore the whole search space beneath it).

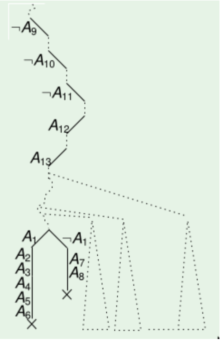


Figure 1: Example of backtracking wasted search-space

4 OBDD - Ordered Binary Decision Diagrams

The OBDD is a canonical representation of boolean formulas. It works as a binary directed acyclic graph, where each node branches setting a variable as $\top$ or $\bot$.
The order of the variables is set a priori but it is important since it impacts the size of the OBDD.

4.1 Generating OBDD - naive

To generate an OBDD we start from an ordered decision tree (trivial binary tree where each variable is set to both values to explore all configuration).
Then we repeatedly apply two reductions:

- **Shared subnodes**: If a subtree occurs twice or more only keep one instance. All the nodes that pointed to the deleted occurrences now point to this one.
  - We can use hash consing to identify identical subtrees.
- **Redundancies**: nodes with the same left and right children can be eliminated.

Of course this method is extremely expensive, since it has to start from a decision tree.

4.2 Incremental Building

> **if-then-else operator.**
>
> $$ite(\neg\phi, \varphi^\top, \varphi^\bot) = ite(\phi, \varphi^\bot, \varphi^\top)$$
> $$\neg ite\phi, \varphi^\top, \varphi^\bot) = ite(\phi, \neg\varphi^\top, \neg\varphi^\bot)$$
> $$ite(\phi, \varphi_1^\top, \varphi_1^\bot) \text{ op } ite(\phi, \varphi_2^\top, \varphi_2^\bot) = ite(\phi, \varphi_1^\top \text{ op } \varphi_2^\top, \varphi_1^\bot \text{ op } \varphi_2^\bot)$$

> $ite(\phi_1, \varphi_1^\top, \varphi_1^\bot)$ op $ite(\phi_2, \varphi_2^\top, \varphi_2^\bot)$ c'è del nesting che non ho voglia di scrivere.

$$\begin{aligned}
OBDD(\top, \{A_1, \dots, A_n\}) &= 1 \\
OBDD(\bot, \{A_1, \dots, A_n\}) &= 0 \\
OBDD(\varphi, \{A_1, \dots, A_n\}) &= ite(A_1, \\
&\qquad\qquad OBDD(\varphi[A_1|\top], \{A_2, \dots, A_n\}), \\
&\qquad\qquad OBDD(\varphi[A_1|\bot], \{A_2, \dots, A_n\}))
\end{aligned}$$

Vedi le slide, le regole sono eterne.

Key points are:

- OBDD è canonico. Se due formule sono equivalenti, e usi lo stesso ordinamento di variabili, l'OBDD dell'una e dell'altra sono uguali.
- L'ordine delle variabili può fare la differenza tra space complexity lineare e esponenziale.
- OBDD è efficiente perché è bottom up.

5 Modern SAT-solving techniques - CDCL

Conflict-Driven Clause-Learning SAT solvers (CDCL) are non-recursive and avoid the excessive backtracking that slowed PDLL solvers down.
The main strengths are:

- Conflict-Driven clause learning
- random restarts
- smart literal selection
- smart preprocessing
- smart indexing
- incremental calls

5.1 Stack representation of truth assignments

A truth assignment μ can be represented as a special stack partitioned into *decision levels*.
Each decision level contains *one* decision literal and all its *implied literals*.
The implied literals keep track of their *antecedent clause*, that is the clause that caused their unit-propagation.

This representation of a truth assignment is equivalent to an *implication graph*.

5.1.1 Implication graphs

Implication graphs are Directed Acyclic Graphs where

- each node is a literal, and each edge is labeled with a clause $l_a \overset{c}{\mapsto} l_b$.
- Decision literals have no incoming edges.
- All edges incoming into the same node l must be labeled with the same clause.
- $l_1 \overset{c}{\mapsto} l_2 \overset{c}{\mapsto} \cdots \overset{c}{\mapsto} l \iff c = (\neg l_1 \vee \neg l_2 \vee \cdots \vee l)$
- Conflicts are signaled by the presence of both l and $\neg l$ in the graph.

The intuitive meaning of these rules is that $l_1 \overset{c}{\mapsto} l_2 \overset{c}{\mapsto} \cdots \overset{c}{\mapsto} l$ indicates l has been obtained from $l_1, l_2, \dots$ by unit propagation on c.

NB. The clauses in the chain of implication don't need to be the same.

UIP - Unique implication point An unique implication point is a node l such that *every* path from the last decision to *both* conflict nodes, passes through l.
Trivially the most recent decision node is an UIP.

5.2 CDCL algorithm

Algorithm 2: CDCL solver

```
status := preprocess($\varphi$,$\mu$);
while true do
    while true do
        status := deduce($\varphi$,$\mu$);
        if status==SAT then
            return sat;
        end
        if status==conflict then
            // Backtrack level and conflict set;
            $\langle blevel, \eta \rangle$ := analyze_conflict($\varphi$,$\mu$);
            if blevel==0 then
                return unsat;
            else
                backtrack(blevel,$\varphi$,$\mu$);
            end
        else
            break;
        end
    end
    decide_next_branch($\varphi$,$\mu$);
end
```

- **preprocess(φ, μ)** simplifies φ into an easier equisatisfiable formula, updating μ.
- **decide_next_branch(φ, μ)** chooses a new decision literal from φ according to some heuristic, and adds it to μ
- **deduce(φ, μ)** performs all deterministic assignments (unit-propagations plus others), and updates φ, μ accordingly.
- **analyze_conflict(φ, μ)** Computes the subset η of μ causing the conflict (conflict set), and returns the "wrong-decision" level suggested by η ("0" means that η is entirely assigned at level 0, i.e., a conflict exists even without branching)
- **backtrack(blevel,φ, μ)** undoes the branches up to blevel, and updates φ, μ accordingly

5.3 Learning

We say that the algorithm is "*conflict-driven*" and "*clause-learning*" because whenever a branch fails, we find the conflict set η and add $C \overset{def}{=} \neg\eta$ to the clause set to avoid doing the same mistake again.

We also avoid the PDLL useless backtracking by using η to decide the point where to backtrack.

5.4 Conflict analysis

Some valid criteria are:

- **decision**: C contains only decision literals

Algorithm 3: Conflict analysis

```
$C$ := conflicting clause;
repeat
    resolve $C$ with the antecedent clause of the last unit-propagated literal $l$ in $C$;
until $C$ meets some criteria;
```

- **last UIP**: C contains only one literal assigned at the current decision level and it is the decision literal (the last UIP)
- **first UIP**: C contains only one literal assigned at the current decision level, and it is the first UIP.

The first UIP is the better strategy used in modern solvers.

5.5 Backjumping

The original strategy for backjumping was to backtrack to the most recent branching point such that the stack did not *fully* contain η, and then unit propagate the unassigned literal on the conflict clause.

The modern strategy is to backtrack to the highest branching point such that the stack contains *all but one* literals in η, and then unit propagate the conflict clause like before.

5.6 Problems with CDCL

One problem with clause learning is that the solver can generate *a lot* of learned clauses. The solution to this is to remove clauses that are not being used for propagation in a while.
A more "lazy" approach is to wait for the clauses to have become too many and to remove the least used ones.

5.7 Random restarts

Another technique implemented in CDCL is to randomly restart the search periodically or when some conditions are met. When restarting the learned clauses are kept, and this action can drastically reduce the search space.

5.8 SAT under assumptions

We can modify a SAT problem by introducing some assumptions $A \overset{def}{=} \{l_1, \dots, l_n\}$ and getting $SAT(\varphi, \{l_1, \dots, l_n\})$.

To implement this variant of SAT it is sufficient to put the assumptions as decisions made before decision level 0. This means that whenever the backtrack tries to jump back to one of those, it hits 0 and returns unSAT.

A strong use case for this is to verify the satisfiability of the same formula φ repeatedly under different assumptions.

> **property.** If the *decision* strategy for conflict analysis is used, then η is the subset of assumptions causing the inconsistency.

5.9 Selection of sub-formulas

Given a CNF formula φ ($\varphi = \bigwedge_{i=1}^{n} C_i$) and a set of *selectors* $S_1, \dots, S_n$ (fresh boolean atoms), we can use the selectors to "toggle" some clauses so that they may contribute or not to the SAT problem at hand.

The selections are set as *assumptions* and φ is modified to make them interact with the rest of the clauses.

5.10 Incremental SAT solving

Many modern sat solvers allow to push and pop subformulas onto a base (possibly empty) formula and check the satisfiability at every step.
By maintaining the state and the learned clauses from the previous steps we can drastically reduce the search space of each subsequent call.

6 First-Order Logic

Giunchiglia

6.1 Theories

In formal logic a *theory* is **a set of axioms**, where each axiom is a FOL closed formula. This is typical used to give a practical *interpretation* to the symbols in the logic.

Another, more practical definition of a theory is **a set of models** constraining the interpretation of the logic.

6.2 Satisfiability Modulo Theories - SMT

SMT is the problem of deciding if a formula is satisfiable under the constraint of some theory or set of theories.

The theories we're most concerned with are:

- Equality and uninterpreted functions
- Difference logic
- Linear arithmetic over the rationals (LRA)
- Linear arithmetic over the integers (LIA)
- Arrays
- Bit Vectors
- Non-linear arithmetic over the reals

6.3 SMT Solvers

We can solve problems of Satisfiability Modulo Theory by expanding on the CDCL solver we've seen already. The high level workflow is:

1. Theory atoms are substituted with fresh boolean variables.
2. A CDCL solver is used to enumerate truth assignments μ_i on the modified formula
3. The set of theory atoms is extracted from the assignment
4. A *theory specific solver* checks only the satisfiability of the theory atoms
5. If they're sat, good, otherwise return to 2

For this method a couple important things are

- The interleaving between CDCL solver and Theory solver
- The Theory solver must offer incremental calling
- The $\mathcal{T}$-solver should provide good conflict sets
- Semantically equivalent but syntactically different clauses should be merged into one
- Early pruning

EUF - Equality and Uninterpreted Functions

Solvers for EUF are based on merge-sets and E-Graphs.

> **E-Graphs.** Given the set of terms occurring in the formula represented as nodes in a DAG (or term bank):
>
> - if $(t = s)$ then merge the eq classes of t and s
> - if $\forall i \in 1...k.class(t_i) = class(s_i)$ then merge the eq classes of $f(t_1, ..., t_k)$ and $f(s_1, ..., s_k)$
> - if $t \neq s$ and t and s belong to the same class, conflict

DL - Difference logic

For difference logic we use a variant of the Bellman-Ford algorithm, where conflicts are revealed by negative cycles.

LRA - Linear Arithmetic over the rationals

For this a variant of the simplex LP algorithm is used. No idea what that is.

LIA - Linear arithmetic over the integers

This is an NP-complete problem, and is tackled with a mix of various techniques like simples, branchbound, and cutting planes.

AR - Arrays

This also is a NP-complete problem.
Solvers usually use EUF combined with the array axioms:

1. $\forall a.\forall i.\forall e(read(write(a, i, e), i) = e)$
   If I write e to a location and then read that same location, I should get e
2. $\forall a.\forall i.\forall j.\forall e.((i \neq j) \rightarrow read(write(a, i, e), j) = read(a, j))$
   If I write in a location i, the values in the rest of the array don't change
3. $\forall a.\forall b.(\forall i.read(a, i) = read(b, i)) \rightarrow (a = b))$
   If all the elements in two arrays are equal, the arrays are equal

7 Kripke Models

The semantic framework for a variety of logics like Modal Logics, Description Logics, and *Temporal Logics*.

The **practical role** of a Kripke model is to describe *reactive systems*. This means nonterminating systems with infinite behaviors (like communication protocols and circuits).

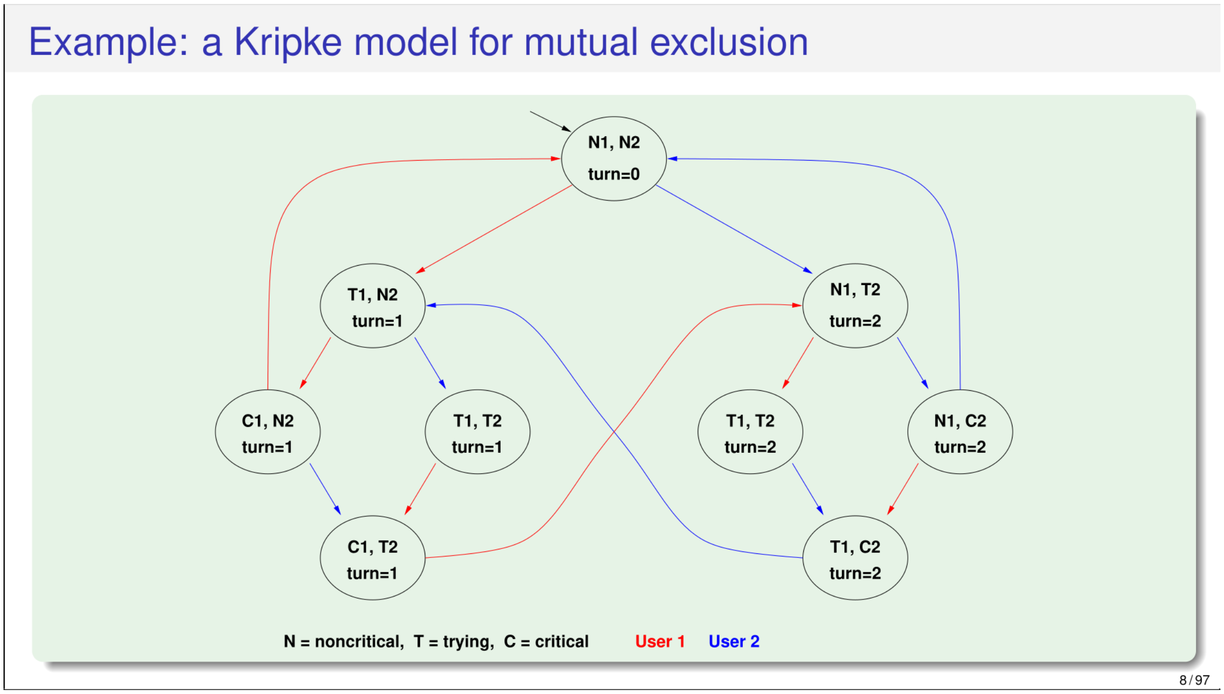
7.1 Formal Definition

A Kripke model $\langle S, I, R, AP, L\rangle$ consists of:

- a finite set of states S
- a set of initial states $I \subseteq S$
- a set of transitions $R \subseteq S \times S$
- a set of atomic propositions AP
- a labeling function $L : S \mapsto 2^{AP}$

We assume R to be total, so for every state s there exists at least one state s' such that $s, s' \in R$. Sometimes we use variables with discrete bounded values $v_i \in \{d_1, \dots, d_k\}$

> **Remark.** Unlike with other types of automata, in Kripke models the values of all variables are always assigned in each state



Example: a Kripke model for mutual exclusion

Path A path in a Kripke model M is an *infinite* sequence of states $\pi = s_0, s_1, \dots \in S^\omega$

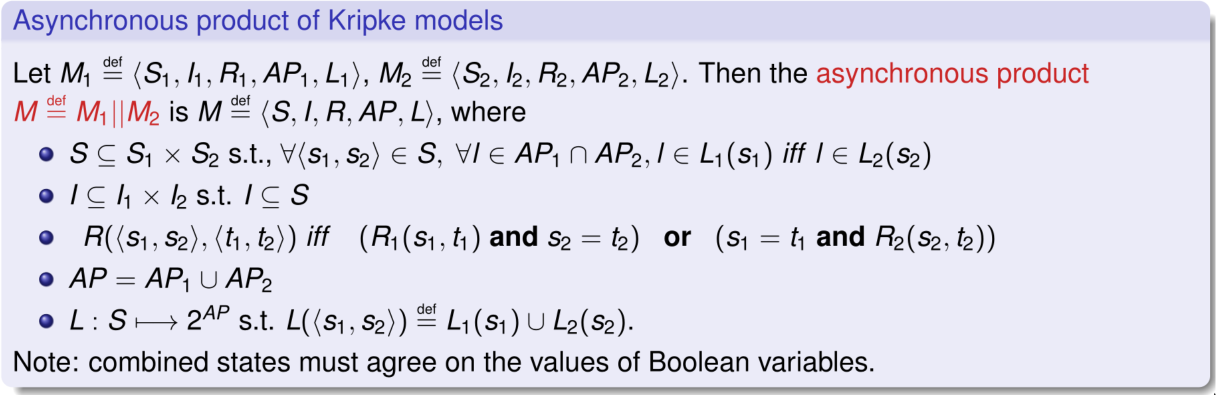
Reachable A state is reachable if there exists a path that includes it

Asynchronous Composition/Product At each time instant, one component is selected to perform a transition

It is a typical formalization for protocols since it models agents well.

$M \overset{def}{=} M_1 || M_2 \overset{def}{=} \langle S, I, R, AP, L \rangle$

- $S \subseteq s_1 \times s_2$ s.t $\forall \langle s_1, s_2 \rangle \in S, \forall I \in AP_1 \cap AP_2, I \in I_1(s_i)$ iff $I \in L_2(s_2)$
- $I \subseteq$ ...



Asynchronous product of Kripke models

Let $M_1 \overset{\text{def}}{=} \langle S_1, I_1, R_1, AP_1, L_1 \rangle$, $M_2 \overset{\text{def}}{=} \langle S_2, I_2, R_2, AP_2, L_2 \rangle$. Then the asynchronous product $M \overset{\text{def}}{=} M_1 || M_2$ is $M \overset{\text{def}}{=} \langle S, I, R, AP, L \rangle$, where

- $S \subseteq S_1 \times S_2$ s.t., $\forall \langle s_1, s_2 \rangle \in S$, $\forall l \in AP_1 \cap AP_2, l \in L_1(s_1)$ *iff* $l \in L_2(s_2)$
- $I \subseteq I_1 \times I_2$ s.t. $I \subseteq S$
- $R(\langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle)$ *iff* $(R_1(s_1, t_1)$ **and** $s_2 = t_2)$ **or** $(s_1 = t_1$ **and** $R_2(s_2, t_2))$
- $AP = AP_1 \cup AP_2$
- $L : S \longmapsto 2^{AP}$ s.t. $L(\langle s_1, s_2 \rangle) \overset{\text{def}}{=} L_1(s_1) \cup L_2(s_2)$.

Note: combined states must agree on the values of Boolean variables.

Synchronous Composition/Product At each time instant, every component performs a transition.

It is a typical formalization for circuits, since it models nicely the behaviour of a clock.

Synchronous product of Kripke models

Let $M_1 \overset{\text{def}}{=} \langle S_1, I_1, R_1, AP_1, L_1 \rangle$, $M_2 \overset{\text{def}}{=} \langle S_2, I_2, R_2, AP_2, L_2 \rangle$. Then the synchronous product $M \overset{\text{def}}{=} M_1 \times M_2$ is $M \overset{\text{def}}{=} \langle S, I, R, AP, L \rangle$, where

- $S \subseteq S_1 \times S_2$ s.t., $\forall \langle s_1, s_2 \rangle \in S$, $\forall l \in AP_1 \cap AP_2, l \in L_1(s_1)$ *iff* $l \in L_2(s_2)$
- $I \subseteq I_1 \times I_2$ s.t. $I \subseteq S$
- $R(\langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle)$ *iff* $(R_1(s_1, t_1)$ **and** $R_2(s_2, t_2))$
- $AP = AP_1 \cup AP_2$
- $L : S \longmapsto 2^{AP}$ s.t. $L(\langle s_1, s_2 \rangle) \overset{\text{def}}{=} L_1(s_1) \cup L_2(s_2)$.

Note: combined states must agree on the values of Boolean variables.

7.2 Descriptor languages

most often a Kripke model is not given explicitly but represented in a structured language (like SMV, VHDL, etc...)

7.2.1 The SMV Language

> **riprendere.**

7.3 Standard Programming Languages

Standard programming languages can be seen as a transition relation in terms also of the program counter.

7.4 Properties

Safety Properties Bad events never happen (deadlock and other bad conditions). This can be seen as imposing that no reachable state satisfies a "bad" condition (e.g. never two processes in critical section)

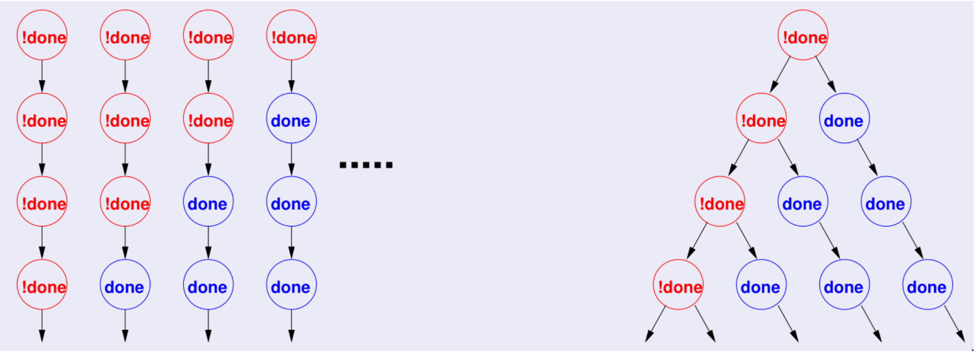
This property can be refuted by a finite behaviour (we just need to prove *one* bad execution). This is fairly obvious.

Liveness Properties Something desirable will eventually happen. This can be refuted by *infinite* behaviour.
Since we're working with finite machines, infinite behaviours are only achieved by loops. Hence they can be detected with (advanced) loop detection.

Fairness Properties Something desirable will happen *infinitely often*. This can be seed as a further restriction on the liveness property, since we no longer require that something happens but we require that it also *keeps* happening.

7.5 Computation trees vs paths

Given a Kripke structure it's execution can be seen as an infinite set of computation paths or an infinite computations tree



8 Temporal Logics

They are divided in *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)*

8.1 LTL - Linear Temporal Logic

An atomic proposition is a LTL formula.
Given φ_1, φ_2 LTL, $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \dots$ are LTL.
$X\varphi_1, G\varphi_1, F\varphi_1, \varphi_2 U \varphi_1, \varphi_2 R \varphi_1$ are new operators.

- Next **X**: X_φ holds in s_t iff φ is true in s_{t+1}
- Finally (or eventually) **F**: Is true if φ will eventually be true down the path.
- Globally **G**: $G\varphi$ is true in s_t iff φ is true in all $s_{t'} \geq s_t$
- Until $\varphi_1 U \varphi_2$: There is a state in which φ_2 is true, and up to that state φ_1 is true
- Release $\varphi_1 R \varphi_2$: φ_2 can become false only if φ_1 does first

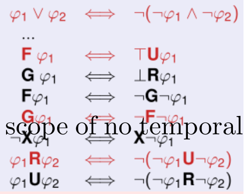
Fairness. **GF**φ is a way to represent fairness, since it indicates that something is gonna happen (F) and that things are always gonna happen (G)

Kripke models These properties are evaluated over paths (infinite, linear sequences of states) so we can go back to the Kripke model's paths. A model M models ϕ if for all paths π in M, $\pi \models \phi$.

NB. $M \not\models \not\Longrightarrow M \models \neg\phi$

8.1.1 Negation Properties

From this properties we can see that we only really need $\wedge, \neg, X, U$ to simulate every other connector



$$\begin{array}{lcl} \varphi_1 \vee \varphi_2 & \Longleftrightarrow & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \dots & & \\ \mathbf{F}\,\varphi_1 & \Longleftrightarrow & \top\mathbf{U}\varphi_1 \\ \mathbf{G}\,\varphi_1 & \Longleftrightarrow & \bot\mathbf{R}\varphi_1 \\ \mathbf{F}\varphi_1 & \Longleftrightarrow & \neg\mathbf{G}\neg\varphi_1 \\ \mathbf{G}\varphi_1 & \Longleftrightarrow & \neg\mathbf{F}\neg\varphi_1 \\ \neg\mathbf{X}\varphi_1 & \Longleftrightarrow & \mathbf{X}\neg\varphi_1 \\ \varphi_1\mathbf{R}\varphi_2 & \Longleftrightarrow & \neg(\neg\varphi_1\mathbf{U}\neg\varphi_2) \\ \varphi_1\mathbf{U}\varphi_2 & \Longleftrightarrow & \neg(\neg\varphi_1\mathbf{R}\neg\varphi_2) \end{array}$$

8.1.2 LTL Tableaux rules

We say that a temporal subformula occurs at top level in φ if it occurs in φ under the scope of no temporal operator

NB. $M \models \varphi$ where φ does not contain temporal operators, φ is only evaluated at the **initial state**.

8.2 CTL - Computation Tree Logic

As in LTL, we keep the boolean operators and extend them with new ones.

- $AX\varphi$

- $A(\varphi U \varphi)$
- $AG\varphi$
- $AF\varphi$
- $EX\varphi$
- $E(\varphi U \varphi)$
- $EG\varphi$
- $EF\varphi$
- $E(\varphi U \varphi)$

The new operators work as "extensions" of the LTL ones:

- **AX**: Necessary next
- **EX**: possible next
- **AF**: necessarily in the future
- **EF**: possible in the future
- **AG**: globally
- **EG**: possible henceforth
- **AU**: necessarily until
- **EU**: possible Until

The **A** can be seen as "all", so the property holds in all branches.
The **E** can be seen as "exists", so there is at least one branch where the property holds.

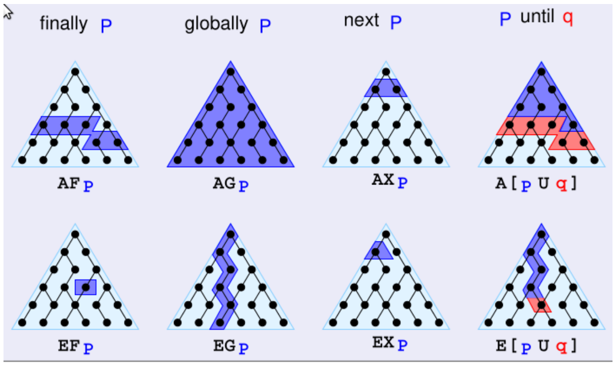
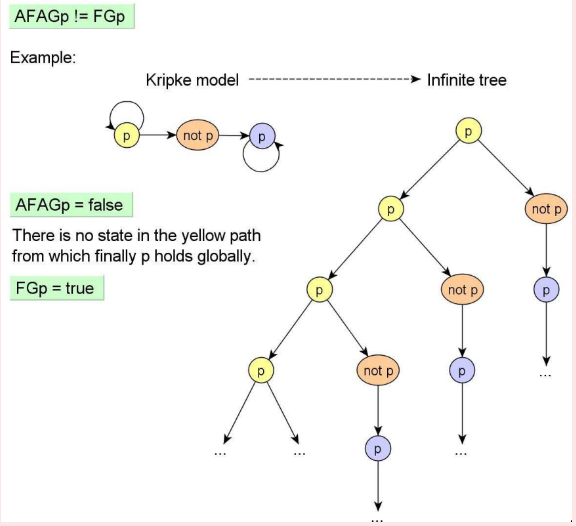


Figure 2: Visual representation of the operators

Watch out!. LTL and CTL are **NOT** equivalent and none of them is strictly more expressive than the other.
There are LTL formulas that cannot be expressed in CTL and vice versa.



AFAGp != FGp

Example:

Kripke model → Infinite tree

AFAGp = false

There is no state in the yellow path from which finally p holds globally.

FGp = true

There are obviously some formulas that can be expressed in both.

8.3 CTL*

CTL* divides formulas into *state formulae* and *path formulae.*

State formulas boolean operations and boolean connections of smaller state formulas are state formulas. If ψ is a *path* formula, $A\psi$ and $E\psi$ are state formulas.

Path formulas State formulas are path formulas. Boolean combinations of path formulas are path formulas. Path/time quantifiers on path formulas are path formulas.

CTL* subsumes both LTL and CTL.

- φ in CTL $\implies$ φ in CTL*
- φ in LTL $\implies$ Aφ in CTL*
- $LTL \cup CTL \subset CTL*$

So CTL* gives us the tools to compare CTL and LTL in one framework.

9 Automata-Theoretic LTL Reasoning

9.1 Infinite word languages

Given an alphabet Σ a ω-word α over Σ is an infinite sequence of elements from Σ.
The set of all infinite words is Σ^ω
A Ω-language L is a collection of ω-words $L \subseteq \Sigma^\omega$.

9.2 Omega-Automata

We consider automaton running over infinite words. A word is *accepted* if there is at least one state which is repeated *infinitely* many times.

9.3 Buchi automata

A non-deterministic Buchi automaton (NBA) is $(Q, \Sigma, \delta, I, F)$

- Q finite set of states
- Σ finite alphabet

- $I \subseteq Q$ set of initial states
- $F \subseteq Q$ set of accepting states
- $\delta \subset Q \times \Sigma \times Q$ transition relation

A deterministic Buchi automaton (DBA) has the same definition except δ is a function instead of a relation $\delta : Q \times \Sigma \mapsto Q$

Runs and Languages A run ρ of A on ω-word $\alpha = a_0, a_1, ...$ is an infinite sequence $\rho = q_0, q_1, ...$ st $q_0 \in I$ and $q_i \overset{a_i}{\to} q_{i+1}$ for $0 \leq i$.

A run ρ is *accepting* if $inf(\rho) \cap F \neq \emptyset$.

The *language* accepted by A is the set of all α accepted by A.

DBA vs NBA DBAs are strictly **less** powerful than NBAs.

9.4 Operations on Buchi Automata

For the NBAs A_1, A_2 we can construct the NBAs for the union of their languages and for their intersection. It is also possible to do complementation but it's hard and out of the scope of this course.

Union The size of the union of two automata is $|A| = |A_1| + |A_2|$.
To build it we just take the two graphs and consider them together. A graph does not need to be connected, so we can leave them like that.

Intersection The size of the intersection of two automata is $|A| \leq |A_1| \cdot |A_2| \cdot 2$.
To build the intersection we consider the combination of each state and we add a flag $Q = Q_1 \times Q_2 \times \{1, 2\}$. The initial states are $I = I_1 \times I_2 \times \{1\}$ and the final ones are $F = F_1 \times F_2 \times \{1\}$.

9.5 LTL and Buchi

validity and satisfiability Let ψ be an LTL formula ($\models \psi$, $\neg\psi$ unsat, $L(A_{\neg\psi}) = \emptyset$).
$A_{\neg\psi}$ is a Buchi Automatons which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ).

Entailment Let φ, ψ LTL formulas. ($\varphi \models \psi$, $\models \varphi \rightarrow \psi$
blah blah

LTL model checking

9.6 Language emptiness checking

To check that a language is *not empty* we just need to find and accepting cycle reachable from an initial state. If we find a loop we know we can follow it forever and touch the final state infinite many times.

9.6.1 Naive Double Nested DFS algorithm

The easy approach to find such loops is to run two nested DFS over the graph. Whenever the outer DFS finds a final state f, the inner one is started. If the inner DFS gets back to f, there is a loop.

This method is $O(n^2)$ and is not very good.

9.6.2 SCC-based algorithm

We can use the *Strongly Connected Components* of a graph to make the naive implementation run in linear time.

9.6.3 Smart Double Nested DFS

The naive method can be improved drastically with two modifications:

1. The inner DFS is not started when a final state is encountered, but only when it is popped.
2. Keep two hash tables and two stacks.

The algorithm is explained better in the slides, the complexity becomes $O(n)$.

9.7 Kripke Models to Buchi Automata

We can transform a Kripke model $M = \langle S, S_0, R, L, AP\langle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F\rangle$ such that

- $Q := S \cup \{init\}$, $init$ being a new initial state
- $\Sigma := 2^{AP}$ total truth-assignments as alphabet symbols
- $I := \{init\}$
- $F := Q$
- $\begin{array}{ll} \delta : & q \xrightarrow{a} q' \iff (q, q') \in R \text{ and } L(q') = a \\ & init \xrightarrow{a} q \iff q \in S_0 \text{ and } L(q) = a \end{array}$

The labels are effectively moved from the nodes to the incoming edges of said nodes.

9.8 From LTL formulas to Buchi Automata

LTL Negative Normal Form Every LTL formula can be converted into its negative normal form which only uses $\wedge, \vee, X, U, R$ on propositional literals.

> **Reminder.** A property of NNF is that every non atom in φ occurs positively.

On-the-fly construction of A_φ

1. Apply the tableau expansion rules to φ

$$\begin{aligned} \psi_1 U \psi_2 &\implies \psi_2 \vee (\psi_1 \wedge X(\psi_1 U \psi_2))[\text{ and } F\psi \implies \psi \vee XF\psi] \\ \psi_1 R \psi_2 &\implies \psi_2 \wedge (\psi_1 \vee X(\psi_1 R \psi_2))[\text{ and } G\psi \implies \psi \wedge XG\psi] \end{aligned}$$

2. Convert all formulas into *disjunctive normal form*
   This is the counterpart to the conjunctive normal form, where $\wedge$ and $\vee$ are inverted
3. huh?