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
- $\alpha \models \beta \iff \alpha \to \beta$ 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 $\varphi_1 \to \varphi_2$ 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;
φ is a set of clauses;
`//Search for a` *`refutation`* `of` φ;
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

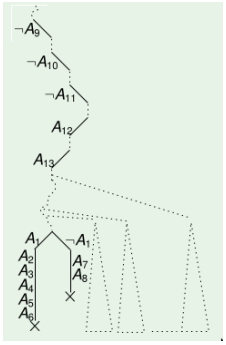


Figure 1: Example of backtracking wasted search-space

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).

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.

$$
\begin{aligned}
ite(\neg\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \varphi^{\bot}, \varphi^{\top}) \\
\neg ite\phi, \varphi^{\top}, \varphi^{\bot}) &= ite(\phi, \neg\varphi^{\top}, \neg\varphi^{\bot}) \\
ite(\phi, \varphi_1^{\top}, \varphi_1^{\bot}) \text{ op } ite(\phi, \varphi_2^{\top}, \varphi_2^{\bot}) &= ite(\phi, \varphi_1^{\top} \text{ op } \varphi_2^{\top}, \varphi_1^{\bot} \text{ op } \varphi_2^{\bot})
\end{aligned}
$$

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

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

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 $l_1, l_2, \ldots$ 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

- **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

Algorithm 2: CDCL solver

```
status := preprocess(φ,μ);
while true do
    while true do
        status := deduce(φ,μ);
        if status==SAT then
            return sat;
        end
        if status==conflict then
            // Backtrack level and conflict set;
            ⟨blevel, η⟩ := analyze_conflict(φ,μ);
            if blevel==0 then
                return unsat;
            else
                backtrack(blevel,φ,μ);
            end
        else
            break;
        end
    end
    decide_next_branch(φ,μ);
end
```

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 $C \stackrel{def}{=} \neg\eta$ 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

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;
```

Some valid criteria are:

- **decision**: C contains only decision literals
- **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 $A \stackrel{def}{=} \{l_1, \ldots, l_n\}$ and getting $SAT(\varphi, \{l_1, \ldots, l_n\})$.

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* $S_1, \ldots, S_n$ (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 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).

6.1 Formal Definition

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

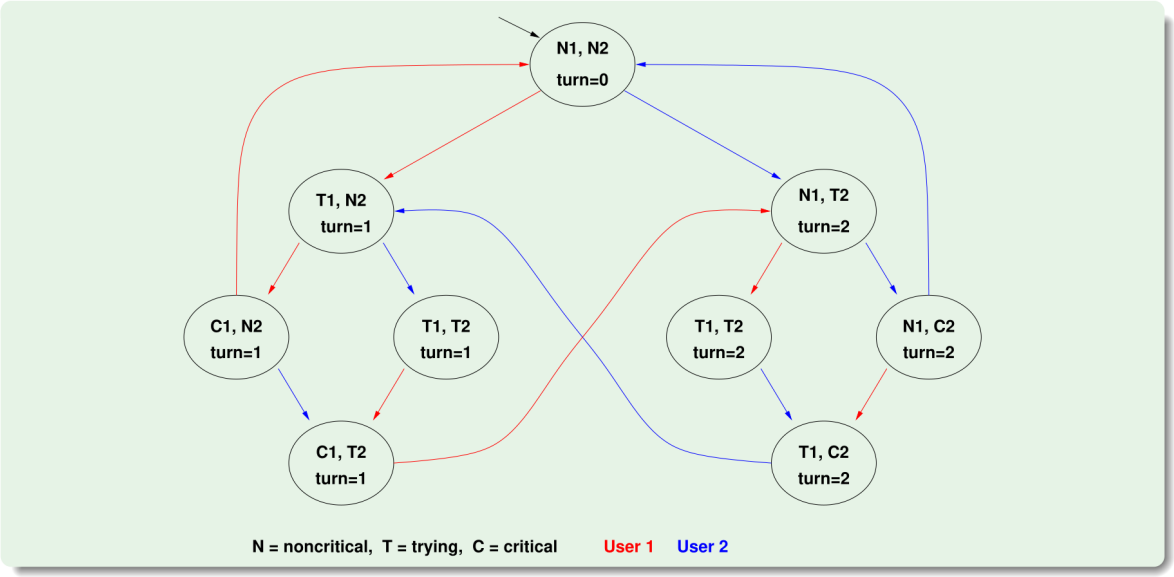
- a finite set of states S
- a set of initial states $I \subseteq S$
- a set of transitions $R \subseteq S \times S$

- 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 $v_i \in \{d_1, \ldots, d_k\}$

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



8 / 97

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

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.

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

- $S \subseteq s_1 \times s_2$ 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 $I \in L_2(s_2)$
- $I \subseteq \ldots$

Asynchronous product of Kripke models

Let $M_1 \stackrel{\text{def}}{=} \langle S_1, I_1, R_1, AP_1, L_1 \rangle$, $M_2 \stackrel{\text{def}}{=} \langle S_2, I_2, R_2, AP_2, L_2 \rangle$. Then the asynchronous product $M \stackrel{\text{def}}{=} M_1 || M_2$ is $M \stackrel{\text{def}}{=} \langle S, I, R, AP, L \rangle$, 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. $L(\langle s_1, s_2 \rangle) \stackrel{\text{def}}{=} L_1(s_1) \cup L_2(s_2)$.

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 $M_1 \stackrel{\text{def}}{=} \langle S_1, I_1, R_1, AP_1, L_1 \rangle$, $M_2 \stackrel{\text{def}}{=} \langle S_2, I_2, R_2, AP_2, L_2 \rangle$. Then the synchronous product $M \stackrel{\text{def}}{=} M_1 \times M_2$ is $M \stackrel{\text{def}}{=} \langle S, I, R, AP, L \rangle$, 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. $L(\langle s_1, s_2 \rangle) \stackrel{\text{def}}{=} L_1(s_1) \cup L_2(s_2)$.

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

6.2 Descriptor languages

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

6.2.1 The SMV Language

riprendere.

6.3 Standard Programming Languages

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

6.4 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.

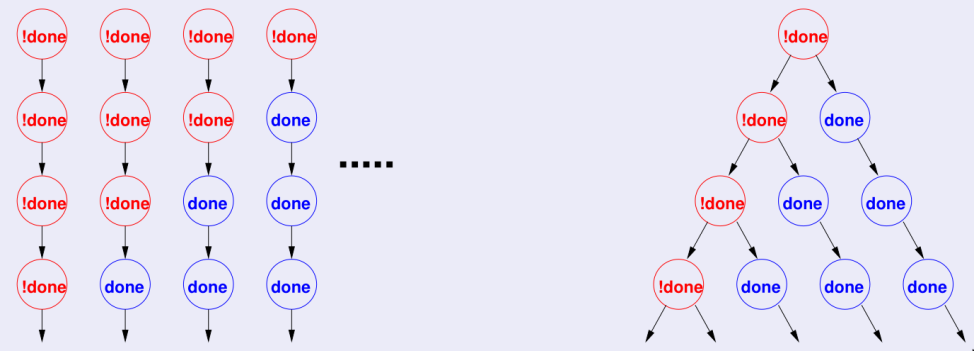
Liveness Properties Something desirable will eventually happen. This can be refuted by *infinite* behaviour.

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.

6.5 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



7 Temporal Logics

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