Formal Methods of Computer Science - Summary

SS2023

Manuel Waibel

October 31, 2023

Contents

1 Computability and Complexity

Definition: Problem

A problem is a question together with an (countably) infinite set of possible instances.
A problem is a decision problem if the question has a yes/no answer.

1.1 Decidability

Definition: Decidability

A decision problem $\mathcal{P}$ is called decidable if there exists an algorithm for $\mathcal{P}$. Otherwise, if there doesn't exits an algorithm for $\mathcal{P}$, then $\mathcal{P}$ is called undecidable.

Definition: Semi-Decidability

A decision problem $\mathcal{P}$ is called semi-decidable, if we can build a program Π such that:

- Π takes as input instances I of $\mathcal{P}$
- if I is a "yes" instance, then Π returns *true*
- if I is a "no" instance, then Π returns *false* or does not terminate

(Π works correctly on all positive instances of $\mathcal{P}$, but may not terminate on the negative instances of $\mathcal{P}$)

If $\mathcal{P}$ is decidable $\implies$ semi-decidable
If $\mathcal{P}$ is semi-decidable $\not\Longrightarrow$ decidable
If $\mathcal{P}$ is decidable $\implies$ $Co-\mathcal{P}$ is decidable
If $\mathcal{P}$ is semi-decidable **and** $Co-\mathcal{P}$ is semi-decidable $\implies$ $\mathcal{P}$ is decidable
If $\mathcal{P}$ is semi-decidable $\not\Longrightarrow$ $Co-\mathcal{P}$ is semi-decidable

1.2 Complexity of decidable Problems

1.2.1 The complexity class *P*:

The class **P** is the collection of all problems that can be solved in polynomial time in the size of the instance.

Definition: P

P consists of all decision problems $\mathcal{P}$ satisfying the following:

- there is a program Π that decides $\mathcal{P}$ and Π is such that
- for all instances I of $\mathcal{P}$, the run time of Π on I is polynomial in $|I|$, i.e. the run time is $O\left(|I|^k\right)$, where k is a constant.

Example:
Model-checking algorithm: Check what is the output of a given formula under a given truth assignment

1.2.2 The complexity class *NP*

NP = Non-deterministic Polynomial time.

"Non-deterministic" here means, having to possibly check all instances. If ones gets "lucky" one can find a solution in polynomial time (being lucky can be enforced by non-determinism).

Example:
Polynomial model-checking (given formula and truth assignment, check output) $\rightarrow$ NP-SAT problem (given a formula - "is there" a truth assignment, which makes the formula true?)

Certificates:
A positive instance for a problem.
SAT: If a formula φ is satisfiable, there exists a certificate.
If the formula φ is unsatisfiable, a certificate cannot be found.

Certificate relation:
A collection of all instances (e.g. formulas in SAT), together with their certificate (in SAT: assignments, which make the formula true).

Definition:

Assume a binary relation R

We say R is polynomially decidable if there is a polynomial-time algorithm that checks given a pair v_1, v_2 of objects (e.g. formula and variable assignment in SAT), wether $(v_1, v_2) \in R$.

We say R is polynomially balanced if $(v_1, v_2) \in R$ implies $|v_2| \leq |v_1|^k$ for some fixed $k \geq 1$.
($=$ "a certificate is bounded within an instance" - e.g. a truth assignment in SAT of a formula is obviously bounded within the size of the formula)

Definition: NP

A decision problem $\mathcal{P}$ is in the class NP if there exists a *polynomially balanced* and *polynomially decidable* certificate relation for $\mathcal{P}$.

Problems in NP can be verified in polynomial time, but there does not (yet) exist a algorithm, which solves them in polynomial time.

Proof: SAT $\in$ NP
$R = \{(\varphi, \mu) \mid \text{formula } \varphi \text{ evaluates to } \textit{true} \text{ under assignment } \mu\}$

- R is a certificate relation by construction: φ is a positive instance of SAT $\Leftrightarrow$ there exists an assignment μ that makes φ evaluate to *true* $\Leftrightarrow (\varphi, \mu) \in R$.
- R is polynomially balanced because each assignment μ for φ can be represented as a subset of variables in φ.
- R is polynomially decidable, because evaluating a propositional formula φ under μ takes only polynomial time.

1.3 Reductions

1.3.1 Types of Reductions

Turing Reductions
The algorithm for a new problem A uses the algorithm for a problem B as a subroutine.

Many-one Reductions
Define a function R, which relates instances of problem A to instances of problem B. All positive instances from A are mapped to the positive instances of B and all negative instances of A are mapped to the negative instances of B.

Proofing the correctness of a reduction in NP, if SAT is involved:

1. Positive instance
2. Certificate
3. Construction of a truth assignment
4. Show that it is a satisfing truth assignment

1.4 NP-completeness

Notation: $\mathcal{P} \leq_R \mathcal{P}'$ "$\mathcal{P}$ can be reduced to $\mathcal{P}'$"

NP-hard:
A problem $\mathcal{P}$ is called NP-hard, if any problem $\mathcal{P}' \in NP$ can be reduced to $\mathcal{P}$ (so $\mathcal{P}' \leq_r \mathcal{P}$).

NP-complete:
A problem $\mathcal{P}$ is called NP-cpmplete, if any problem $\mathcal{P}' \in NP$ can be reduced to $\mathcal{P}$ (NP-hard) **and** $\mathcal{P}$ itself is also in NP.

Be mindful for implications on the hard-/completness of "special" problems.
If the instances are the same, but the questions changes, one cannot directly imply NP-completeness from a NP-complete problem (e.g. 1-IN-3-SAT $\subset$ NAESAT $\subset$ 3-SAT).
If the instances are a proper subset and the question stays the same, one **can** imply NP-hardness (3-SAT from SAT).

1.5 Other complexity classes

1.5.1 Logarithmic Space L

Definition: Class L

L is the clas of all problems that can be solved by a program that uses logarithmic space in respect to an input I. The program can use at most $O(\log_2 |I|)$ bits of read/write memory.

If a problem can be solved in L, it is solvable in polynomial time, because the stored internal states can only be 2^n, where n is the number of bits. So it is automatically polynomially bound.

1.6 Useful NP problems to know

1.6.1 Halting

Instance: A program Π and an input string I
Question: Does the program Π terminate on input I?

Halting is semi-decidable.

1.6.2 Co-Halting

Instance: A program Π and an input string I
Question: Does the program Π not terminate/run forever on input I?

Co-Halting is not decidable.

1.6.3 Correctness

Instance: Source code for a program Π that takes a string I_1 as input and outputs a string I_2.
Question: Does Π return I_2 when run on input I_1?

1.6.4 Reachable-Code

Instance: Source code of a program Π, a number n of a line in Π.
Question: Is there an input I for Π such that the run of Π on I will reach the code on line n?

1.6.5 SAT, 3-SAT, Validity

SAT:
Instance: Propositional formula φ.
Question: Is φ satisfiable?

3-SAT:
Instance: Propositional formula φ in 3-CNF (i.e., CNF where each clause consists of exactly 3 literals).
Question: Is φ satisfiable?

Validity:
Instance: Propositional formula φ.
Question: Is φ valid?

SAT and 3-SAT are NP-complete.

1.6.6 3-Colorability

Instance: Undirected graph $G = (V, E)$.
Question: Does G have a 3-coloring? I.e. an assignment of one of 3 color to each of the vertices in V such that any two vertices i, j connected by an edge $[i, j] \in E$ do not have the same color?

1.6.7 Dominant-Set

Instance: An undirected graph $G = (V, E)$, and an integer k.
Question: Does there exist a dominating set of vertices of size at most k, i.e., is there a set $S \subseteq V$ with $|S| \leq k$ such that for every vertex $v \in V$ it either holds $v \in S$ or there is some $w \in S$ such that $(v, w) \in E$.

1.6.8 Vertex-Cover

Instance: An undirected graph $G = (V, E)$, and an integer k.
Question: Does there exist a vertex cover of vertices of size at most k, i.e. is there a set of vertices $S \subseteq V$ with $|S| \leq k$, such that it holds for every edge $(v, w) \in E$ between two vertices $v \in V$ and $w \in V$ at least either v or w are contained in S.

Vertex cover is NP-hard.

1.6.9 Independent-Set

Instance: An undirected graph $G = (V, E)$, and an integer k.
Question: Does there exist a vertex cover of vertices of size at most k, i.e. is there a set of vertices $S \subseteq V$ with $|S| \leq k$, such that it holds for all vertices $v \in S$ and $w \in S$, that there is no edge $(v, w) \in E$.

2 Satisfiability

The goal of this block is to construct a decision procedure for equality logic with uninterpreted fuction (EUF). Given is usually a EUF-formula φ^{EUF}, which is then step by step reduced to a SAT problem φ^{P}. We can then use SAT-solvers to show the validity of φ^{EUF} or provide a counter example, if the SAT-solver finds a model.
It holds:

φ^{EUF} is E-valid iff φ^{E} is E-valid iff φ^{P} is unsatisfiable.

2.1 Propositional logic

2.1.1 Syntax

- Boolean variables like p, q, r, p_1, ... represent facts

  e.g. r represents "it is raining"

- φ, ψ, ... for formulas
- To combine formulas and/or atoms, we can use conjunction ($\wedge$), disjunction ($\vee$), implication ($\implies$), equivalence ($\leftrightarrow$) and xor ($\oplus$)
- There are also the formulas $\top$ (verum) and $\bot$ (falsum)

2.1.2 Semantics

Truth assignments assign truth values (0 or 1) to variables, but not formulas (we have to evaluate them).
I.e. $I(p) = 1$

Models:
If a truth assignment makes a formula true, we call this assignment a **model** of the formula.

Entailment:
$I \models \varphi$ iff $I(\varphi) = 1$
$I \not\models \varphi$ iff $I(\varphi) = 0$

$W \models \varphi$ iff $Mod(W) \subseteq Mod(\varphi)$, where W is a set of formulas.
To show $\varphi \models \psi$ show $\varphi \implies \psi$ is valid, or further more $\neg(\varphi \implies \psi)$ is unsatisfiable.

The empty conjunction ($\wedge$) is 1 for all interpretations, it is equivalent to $\top$

2.2 Tseitin translation

In a Tseitin translation the formula is decomposed into a tree. Then each node (= atom or subformula) in the tree gets assigned a new atom l_i, which represents the atom or subformula.
Example:
$\varphi : p$ then $l_1 \leftrightarrow p$

The formulas are then brought to CNF.
So $l_1 \leftrightarrow p$ becomes $(\neg l_1 \vee p) \wedge (l_1 \vee \neg p)$

In the exams it is usually the other way around, given a decomposed Tseitin translation, reconstruct the original formula.

Useful translations (from here):

Type	**CNF**
$l = x_1 \wedge x_2$	$(\overline{x_1} \vee \overline{x_2} \vee l) \wedge (x_1 \vee \bar{l}) \wedge (x_2 \vee \bar{l})$
$l = x_1 \vee x_2$	$(x_1 \vee x_2 \vee \bar{l}) \wedge (\overline{x_1} \vee l) \wedge (\overline{x_2} \vee l)$
$l = \overline{x_1 \wedge x_2}$ (NAND)	$(\overline{x_1} \vee \overline{x_2} \vee \bar{l}) \wedge (x_1 \vee l) \wedge (x_2 \vee l)$
$l = x_1 \oplus x_2$ (XOR)	$(\overline{x_1} \vee \overline{x_2} \vee \bar{l}) \wedge (x_1 \vee x_2 \vee \bar{l}) \wedge (x_1 \vee \overline{x_2} \vee l) \wedge (\overline{x_1} \vee x_2 \vee l)$

2.3 Implication graph

2.3.1 Notation

`x = v@d`, means that `x` is assigned to `v` at decision level `d` (`x@d` or ¬`x@d` for short).

Dual of a literal:
$l^d = 1$, if $l = 0$
$l^d = 0$, if $l = 1$

2.3.2 Antecendent

Given a non-unit clause C and a partial assignment σ. Simplify C with σ to get the antecedent of C.

Example:

$$\begin{aligned}
&C : \neg x_1 \vee x_4 \vee \neg x_3 \\
&\sigma = \{x_1 \mapsto 1, x_4 \mapsto 0\} \\
\\
&\neg x_1 \vee x_4 \vee \neg x_3 \\
&\neg 1 \vee 0 \vee \neg x_3 \\
&0 \vee 0 \vee \neg x_3 \\
\\
&\rightarrow \mathit{antecendent}(\neg x_3) = C
\end{aligned}$$

2.3.3 Definition

An implication graph is a labeled DAG $G = (V, E)$, where:

- Each node is labeled as `l@d`
- The edges $E = \{(v_i, v_j) \mid v_i, v_j \in V, v_i^d \in \mathit{Antecendent}(v_j)\}$
- If G has a conflict, we label the node with κ and all incoming edges $\{(v, \kappa) \mid v^d \in c\}$ labeled with clause c

  A conflict is a c that cannot by satisfied, due to the assignments already being defined

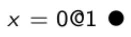
The order of the rules is important! Rules must be evaluated from <u>top to bottom</u>!

2.3.4 Example 1

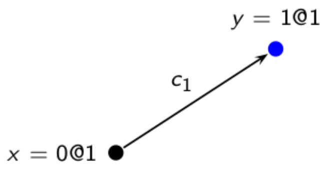
Given:

$$\begin{aligned}
c_1 &: x \vee y \\
c_2 &: x \vee z \\
c_3 &: \neg y \vee \neg z
\end{aligned}$$

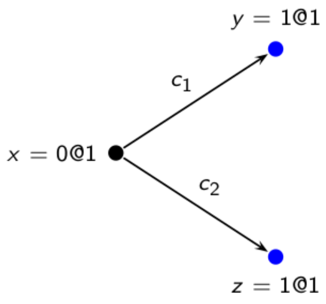
We make a initional decision and set x to 0: `x = 0@1`



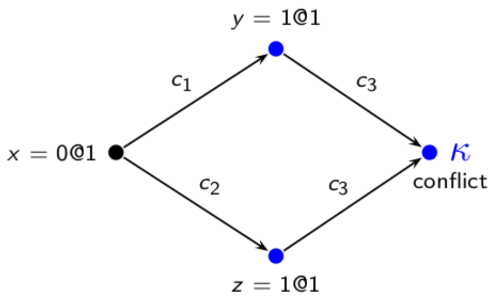
Because we set x to 0 to satisfy c_1 we must set y to 1. We mark the node and the edge accordingly.



Because we set x to 0 to satisfy c_2 we must set z to 1. We mark the node and the edge accordingly.



Because we set y and z to 1 we have a conflict in c_3. We mark this accordingly



2.3.5 Example 2

Given:
Truth assignemnt: $\{\neg x_9@1, \neg x_{10}@3, \neg x_{11}@3, x_{12}@2, x_{13}@2\}$
Current decision assignment: $\{x1@6\}$

$$
\begin{aligned}
c1 &: \neg x_1 \vee x_2 \\
c2 &: \neg x_1 \vee x_3 \vee x_9 \\
c3 &: \neg x_2 \vee \neg x_3 \vee x_4 \\
c4 &: \neg x_4 \vee x_5 \vee x_{10} \\
c5 &: \neg x_4 \vee x_6 \vee x_{11} \\
c6 &: \neg x_5 \vee \neg x_6 \\
c7 &: x_1 \vee x_7 \vee \neg x_{12} \\
c8 &: x_1 \vee x_8 \\
c9 &: \neg x_7 \vee \neg x_8 \vee \neg x_{13}
\end{aligned}
$$

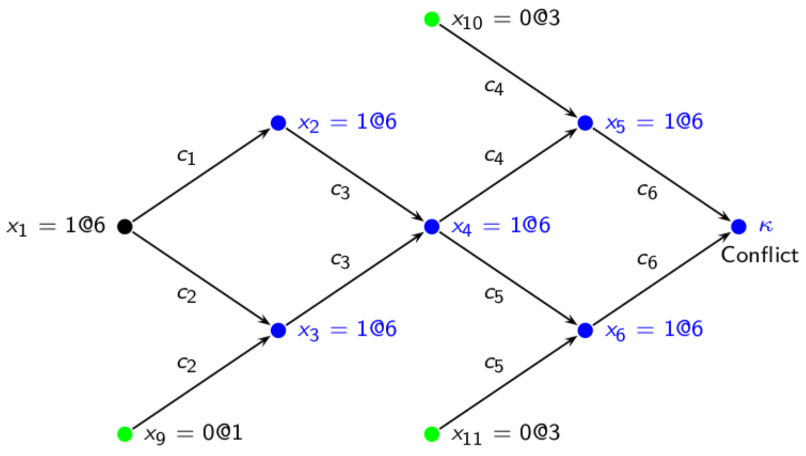


Figure 1: Resulting implication graph

2.3.6 Backtracking

Identify unique inplication points (UIP). UIPs are nodes, where all paths to the conflict node go through.

1. Draw a "fence" around the root nodes (nodes that do not have incoming edges), and include the conflict node κ (see figure 2).
2. Construct a new rule with the nodes outside of the fence

   $x_1@6, \neg x_9@1, \neg x_{10}@3, \neg x_{11}@3 \longrightarrow c_{10} : \neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$

   We backtrack to the "highest" decision level excluded by the fence. In this example it will be level 3 (see figure 3)

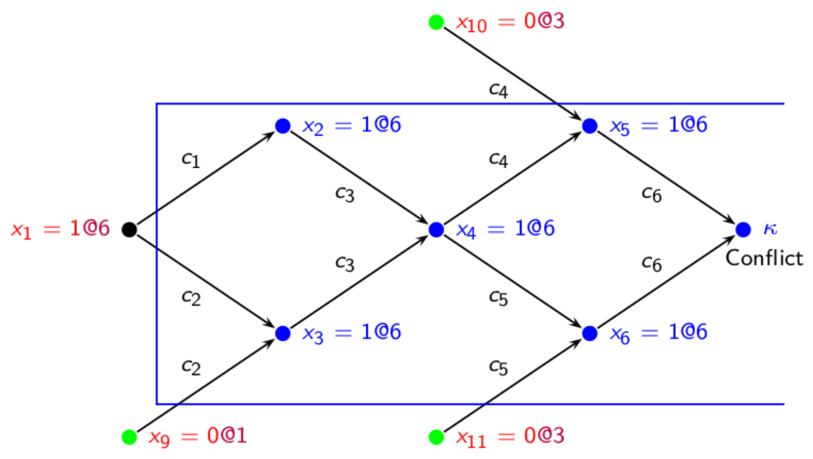


Figure 2: Fence after step 1

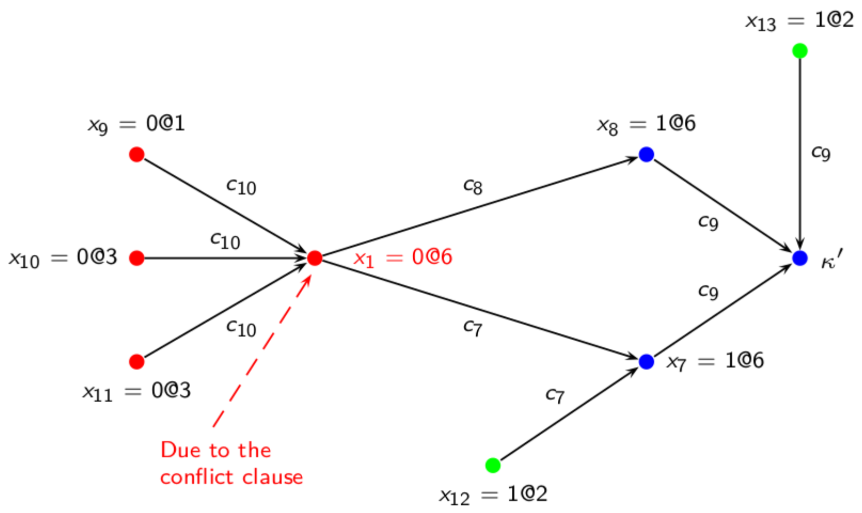


Figure 3: Fence after step 2

2.4 Theory of equality

2.4.1 Rules

Definition of $\doteq$:
The meaning is defined by the axioms of $\mathcal{T}$

1. $\forall x(x \doteq x)$ (reflexivity)
2. $\forall, y((x \doteq y) \implies (y \doteq x))$ (symmetry)
3. $\forall x, y, z((x \doteq y) \wedge (y \doteq z) \implies (x \doteq z))$

Substitution axioms:

$$\forall x_1, y_1, ..., x_n, y_n \left(\bigwedge_{i=1}^{n} x_i \doteq y_i \to f(x_1, ..., x_n) \doteq f(y_1, ..., y_n) \right)$$
$$\forall x_1, y_1, ..., x_n, y_n \left(\bigwedge_{i=1}^{n} x_i \doteq y_i \to p(x_1, ..., x_n) \leftrightarrow p(y_1, ..., y_n) \right)$$

Modus Ponens (MP)

$$\frac{\begin{array}{c}\mathcal{I} \models \Phi \\ \mathcal{I} \models \Phi \to \Psi\end{array}}{\mathcal{I} \models \Psi} \qquad \frac{\begin{array}{c}\mathcal{I} \models \Phi \to \Psi \\ \mathcal{I} \models \Phi\end{array}}{\mathcal{I} \models \Psi}$$

Modus Tolens (MT)

$$\frac{\begin{array}{c}\mathcal{I} \not\models \Psi \\ \mathcal{I} \models \Phi \to \Psi\end{array}}{\mathcal{I} \not\models \Phi} \qquad \frac{\begin{array}{c}\mathcal{I} \models \Phi \to \Psi \\ \mathcal{I} \not\models \Psi\end{array}}{\mathcal{I} \models \Phi}$$

For all the other rules, check the "*sem-argu.pdf*" provided in Tuwel.

2.5 E-Logic and E-Formulas

To translate a formula into an E-Formula Ψ_E:

- We replace all constants c_i with v_{c_i}, where v_{c_i} is a new term variable
- We replace all boolean variables b_i with <u>two</u> new term variables $v_{b_i,1}$ and $v_{b_i,2}$ in the form $v_{b_i,1} \doteq v_{b_i,2}$

2.5.1 Equality graph

1. Add all atoms to the graph
2. Connect them via their corresponding edges

   e.g. a dashed lines for an equality $\doteq$

   and a solid line for disequality $\not\doteq$

There are different types of paths, which can have different meanings for the satisfiability of the formula.

- **Equality path:** An equality path is a path consisting only of edges from $E_{\doteq}$

  An equality path between x and y is denoted by $x \doteq^* y$

- **Disequality path:** A disequality path is a path consisting of edges from $E_{\doteq}$ and exactly one edge from $E_{\neq}$

  a path between x and y is denoted as $x \neq^* y$

- **Simple path:** A simple path is a path without a cycle

Definition: Contradictroy cycle

A contradictory cycle in $G^E(\varphi^E)$ is a cycle with exaclty one disequality edge (A disequality path, that is a cycle).

Theorem: E-unsatisfiability

A subgraph of $G^E(\varphi^E)$ is E-unsatisfiable iff it contains a contradictory cycle.

2.6 Sparse Method - Reduction of E-formulas to propositinoal formulas

For φ^E generate two formulas $e(\varphi^E)$ and B_t such that

$$\varphi^E \text{ is E-satisfiable} \quad \text{iff} \quad e(\varphi^E) \wedge B_t \text{ is satisfiable}$$

$e(\varphi^E)$ is the propositinal skeleton generated as follows:

1. Choose an ordering on variables (and constants)

   e.g. $x_1 < x_2 < x_3 < ...$

2. Orient the equations according to the ordering

3. Replace $x_i \doteq x_j$ by $e_{i,j}$

B_t is a conjunction of transitivity constraints:

1. Construct the equality graph (see 2.5.1)

2. Then draw the non-polar graph $G^E_{NP}(\varphi^E)$ with disregarding the equality type ($\doteq$ is the same as $\neq$)

3. Extract all subgraphs, which are <u>not</u> a triangle (cycle with length 3) and <u>have a cycle</u> with a length of <u>at least</u> 4.[1]
4. Make the subgraphs chordal, by adding edges to transform them into triangle graphs

   This way we now add the transitivity relations
5. Add all constraints of each triangle, by formulating following formulas <u>for each triangle</u> (each triangle adds 3 new formulas)

   $(e_{i,j} \wedge ej, k \implies e_{k,i})\wedge$

   $(e_{i,j} \wedge ek, i \implies e_{j,k})\wedge$

   $(e_{k,i} \wedge ej, k \implies e_{i,j})$

Now to conclude write $e(\varphi^E) \wedge B^t$.

2.7 Ackermann's reduction

To prove validity of an EUF-formula.

1. Number all functions instances from the inside out
   - $F(F(x)) \implies F_2(F_1(x))$
   - The same functions with the same variables inside get replaced by the same new isntances.

     Example: $F(x_1) \doteq F(x_2) \vee F(x_1) \not\doteq F(x_3)$

     becomes

     $F_1(x_1) \doteq F_2(x_2) \vee F_1(x_1) \not\doteq F_3(x_3)$
2. Associate all function instances F_i with f_i (or g_i, ...)

   $$\underbrace{F_2(\overbrace{F_1(x)}^{f_1})}_{f_2}$$
3. Compute $flat^E(\varphi^{EUF})$ by replacing all <u>top-level</u>′ instances of F_i by f_i

   $$\underbrace{F_2(\overbrace{F_1(x)}^{f_1})}_{f_2} \text{ becomes } f_2$$

[1]We do this, because a triangle already means, that the atoms are transitively connected. By only choosing cycles with length > 3 we add actual information to the formula, by formalating transitivity constraints, which are not already part of the formula.

4. Compute the functionality constraints $FC^E(\varphi^{EUF})$

   Let $arg(F_i)$ denote the argument of function F_i

   Write

   $\bigwedge_{i=1}^{m_F-1} \bigwedge_{j=i+1}^{m_F} (arg(F_i) \doteq arg(F_j) \implies f_i \doteq f_j)$

   Example:

   $f_2 \doteq 0$ becomes $(x \doteq f_1 \implies f_1 \doteq f_2) \implies f_2 \doteq 0$

2.8 Additional material from "*Tutorial: Logic and Proof Techniques*"

For some reason this is also sometimes asked in block 2 in the exams, although not explicitly part of block 2 (but taught by prof. Egly). Enjoy.

2.8.1 Noetherian/well-founded induction

The term "well-founded" means, that there is a (lexicographic) order $\sqsubseteq$ in a set, where:

$$(s,t) \sqsubseteq (s',t') \quad \text{iff} \quad \begin{cases} s < s', \\ s = s' \text{ and } t \leq t' \end{cases}$$

So in simple terms, the set is "ordered" (then it is also called a "poset"). The "lexicographic" ordering is defined just like the name suggests for text characters.
For $S \times S$ (combination of elements):
$(1,2) \leq (1,2)$
$(1,2) < (1,3)$
$(1,2) < (2,2)$

The principle of this induction is as follows:

Definition: Noetherian/well-founded indction

Let $(S, \leq)$ be well-founded set and let $\mathcal{P}(x)$ be a statement involving a variable x. Suppose

1. $\mathcal{P}(m)$ is true for each minimal element m of S (base case)
2. for each non-minimal element x, if $\mathcal{P}(y)$ is true $\forall y < x$, then $\mathcal{P}(x)$ is also true (induction step)

Then $\mathcal{P}(x)$ is true for all $x \in S$.

$\forall x \in S\,[(\forall y \in S(y < x \implies \mathcal{P}(y))) \implies \mathcal{P}(x)] \implies \forall z \in S \mathcal{P}(z)$

The steps needed (at least in the exams) are usually the same:

1. Determine a **Base case**
   a) Find the minimal element(s)

      S is usually defined over $\mathbb{N}$ or $\mathbb{N}_0$

      So in the case of $\mathbb{N}$ the minimal element would be 1

      In the case of $\mathbb{N}_0$ the minimal element would be 0

   b) Show that the function returns a correct output for this case
      i. Insert the minimal elements into the function
      ii. Compare the result with the output statement of the function.

          The output statement is always given with the function.

          e.g. "Output: The computed non-negative integer value for x, y"

2. Formulate a **Induction hypothesis**

   Just use exaclty the sentence below:

   "*Pick an arbitrary non-minimal element* (x, y) *and assume that* $\mathcal{P}(x', y')$ *is true for all* $(x', y') \sqsubset (x, y)$."

3. Perform the **Induction step**

   We want to show that $\mathcal{P}(x, y)$ is true for our defined (x, y) from step 2.

   a) Perform a case split for all x and y which are greater or equal to your minimal element (e.g. $x \geq 0$ and $y \geq 0$):
      i. Case 1: $x =$ *[min elem.]*

         Compute output for the function (e.g. $A(0, y) = y + 1$)

         "*Then* $A($*[min elem.]*$, y) = ...$ *and* $\mathcal{P}(0, y)$ *it true*"

      ii. Case 2: $x \neq$ *[min elem.]* $\wedge\, y =$ *[min elem.]* (A is the funtion name in this case)

          "*Then [output]* $\sqsubset (x,$ *[min elem.]*$)$. *By the induction hypothesis* $\mathcal{P}($*[output]*$)$ *is true and* $A(x,$ *[min elem.]*$) = A($*[output]*$)$. *Therefore* $\mathcal{P}(x,$ *[min elem.]*$)$ *holds.*"

      iii. Case 3: $x \neq$ *[min elem.]* $\wedge\, y \neq$ *[min elem.]* (A is the funtion name in this case)

           "*Then ([output(partial output)]). For ([partial output]) it holds that ([partial output])* $\sqsubset (x, y)$ *and therefore* $\mathcal{P}($*[partial output]*$)$ *holds.*
           For $\forall z \in \mathbb{N}_{(0)}, ($*[output]*$, z)$ *it also holds that* $($*output*$, z) \sqsubset (x, y)$ *and therefore* $\mathcal{P}($*output*$, z)$ *holds.*"

4. "*We conclude, that* $\mathcal{P}(x, y)$ *holds for all* $(x, y) \in \mathbb{N}_{(0)} \times \mathbb{N}_{(0)}$" $\square$

3 Deductive Verification of Programs

Pre-condition: Has to/should be fulfilled upon executing the program.
Post-condition: Has to/should be fulfilled upon exiting the program.
Contract of a program: Combination of pre- and postcondition.

3.1 Imperative language - IMP

`Int`	positive and negative (integer) numerals
`Loc`	locations
`AExp`	arithmetic expressions
`BExp`	boolean expressions
`P`	programs

3.1.1 Evaluation

"If F_1 ... F_k, then G":
$$\frac{F_1 \; ... \; F_k}{G}$$

Execution of p in state σ results in state σ':
$\langle p, \sigma \rangle \rightarrow \sigma'$

Evaluation of a in state σ has the resul/value v:
$\langle a, \sigma \rangle \rightarrow v$

Deconstruct the rules and build a derivation tree.

3.2 Hoare logic

Make assertions about IMP programs using axiomatic semantics of IMP $\rightarrow$ Hoare logic

3.2.1 Partial correctness

$$\{A\} \text{ p } \{B\}$$

"For all states, that fulfill requirement A, $\underline{\textbf{if} \text{ the program } p \text{ terminates}}$, then the new state satisfies requirement B."

3.2.2 Total correctess

$$[A] \text{ p } [B]$$

”For all states, that fulfill requirement A, the program p terminates, then the new state satisfies requirement B.”

3.3 Proofing correctness

$\{x = 0\}$ x := x + 1 $\{x = 1\}$ Is it partially correct? $\to$ Yes.

Proof:
We need to prove:
$\sigma \models x = 0 \implies (\forall \sigma'.\ \langle x := x + 1, \sigma\rangle \to \sigma' \implies \sigma' \models x = 1)$

Assume $\sigma \models x = 0$ iff $\langle x, \sigma\rangle \to \sigma(x)$ and $\langle 0, \sigma\rangle \to 0$ and $\sigma(x) = 0$

Now we need to prove:
$(\forall \sigma'.\ \langle x := x + 1, \sigma\rangle \to \sigma' \implies \sigma' \models x = 1)$

$\langle x := x + 1, \sigma\rangle \to$

$$\frac{\langle x + 1, \sigma\rangle \to}{\langle x := x + 1, \sigma\rangle \to}$$

$$\cfrac{\cfrac{\langle x, \sigma\rangle \to 0 \ \langle 1, \sigma\rangle \to 1}{\langle x + 1, \sigma\rangle \to}}{\langle x := x + 1, \sigma\rangle \to}$$

$$\cfrac{\cfrac{\langle x, \sigma\rangle \to 0 \ \langle 1, \sigma\rangle \to 1}{\langle x + 1, \sigma\rangle \to 1}}{\langle x := x + 1, \sigma\rangle \to \sigma(x/1)} \qquad \to \sigma(x/1) = \sigma' \to \sigma' \models x = 1$$

3.4 Hoare triples/Hoare rules

”Triple is provable using Hoare rules”
$\vdash \{A\}$ p $\{B\}$

”Is a valid Hoare triple”
$\models \{A\}$ p $\{B\}$

If $\{A\}$ p $\{B\}$ is probable using Hoare rules, then $\{A\}$ p $\{B\}$ is valid.

3.4.1 Hoare rules for $\vdash \{A\}$ p $\{B\}$

Assignment rule

$$\frac{}{\{B[x/a]\}\ \mathrm{x := a}\ \{B\}}$$

Rule of consequence

$$\frac{A \implies A' \quad \{A'\}\ \mathrm{p}\ \{B'\} \quad B' \implies B}{\{A\}\ \mathrm{p}\ \{B\}}$$

Sequencing rule

$$\frac{\{A\}\ p_1\ \{C\} \quad \{C\}\ p_2\ \{B\}}{\{A\}\ p_1; p_2\ \{B\}}$$

Skip rule

$$\frac{}{\{A\}\ \textbf{skip}\ \{A\}}$$

Abort rule
Undefined state upon aborting (non-terminating state)

$$\frac{}{\{true\}\ \textbf{abort}\ \{B\}}$$

If-then-else rule

$$\frac{\{A \wedge b\}\ p_1\ \{B\} \qquad \{A \wedge \neg b\}\ p_2\ \{B\}}{\{A\}\ \textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \{B\}}$$

While loops

$$\frac{\{A \wedge b\}\ \mathrm{p}\ \{A\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ p\ \textbf{od}\ \{A \wedge \neg b\}}$$

A loop invariant A, holds after each iteration of the loop.

An inductive loop invariant A, holds before the first iteration of the loop and before and after each iteration of the loop.
That is $\{A \wedge b\}$ p $\{A\}$ (this is what one needs to prove to prove inductiveness).

Example for (inductive) loop invariants:
Given program:

```
x := 0; y := 0; n := 10;
while x < n do
    x := x + 1; y := y + x;
od
```

- $x \leq n$: inductive invariant (also an invariant)
- $x < n$: not an invariant and also not an inductive invariant
- $y \geq 0$: invariant (but not an inductive invariant)

3.4.2 Using Hoare rules

Rule for Assignment
When $\{A\}$ x := a $\{B\}$ then use $\vdash\{B[x/a]\}$ x := a $\{B\}$

Examples: Are following Hoare triples provable?

$\{y = 4\}$ x := 4 $\{y = x\}$	replace values: $\{(y = x)[x/4]\}$ x := 4 $\{y = x\}$ $\rightarrow \{y = 4\}$ x := 4 $\{y = x\}$ ✓
$\{x + 1 = y\}$ x := x + 1 $\{x = y\}$	$\{(x = y)[x + 1]\}$ x := x + 1 $\{x = y\}$ $\rightarrow \{x + 1 = y\}$ x := x + 1 $\{x = y\}$ ✓
$\{y = x\}$ y := 0 $\{y = x\}$	$\{(y = x)[y/0]\}$ y := 0 $\{y = x\}$ $\rightarrow \{0 = x\}$ y := 0 $\{y = x\}$ $\oslash$
$\{z = x\}$ y := x $\{z = x\}$	Precondition is equal to postcondition without changing z or x ✓
$\{\forall y.x = x\}$ y := x $\{\forall y.y = x\}$	The y in the pre- and postcondition are not the same, as the y in the program. We can rewrite the triple as follows: $\{\forall z.z = x\}$ y := x $\{\forall z.z = x\}$ This triple is valid, since the precondition can be derived from the postcondition. ✓

Steps for proving $\{A\}$ while b do p od $\{B\}$:

1. Think of a <u>inductive invariant</u> I which satisfies $\{I \wedge b\}$ p $\{I\}$
2. $\dfrac{\{I \wedge b\}\ \text{p}\ \{I\}}{\{I\}\ \textbf{while}\ \text{b}\ \textbf{do}\ \text{p}\ \textbf{od}\ \{I \wedge \neg b\}}$
3. Show that $A \implies I$ for $\{I\}$ **while** b **do** p **od** $\{I \wedge \neg b\}$ by rules of implication
4. Show $I \wedge \neg b \implies B$ by rules of implication

3.4.3 Hoare rules for $\vdash[A]$ p $[B]$

Assignment rule

$$\frac{}{[B[x/a]]\ \text{x := a}\ [B]}$$

Rule of consequence

$$\frac{A \implies A'\quad [A']\ \text{p}\ [B']\quad B' \implies B}{[A]\ \text{p}\ [B]}$$

Sequencing rule

$$\frac{[A]\ p_1\ [C]\ [C]\ p_2\ [B]}{[A]\ p_1; p_2\ [B]}$$

Skip rule

$$\frac{}{[A]\ \textbf{skip}\ [A]}$$

Abort rule
Undefined state upon aborting (non-terminating state)

$$\frac{}{[\color{red}{false}]\ \textbf{abort}\ [B]}$$

If-then-else rule

$$\frac{[A \wedge b]\ p_1\ [B] \qquad [A \wedge \neg b]\ p_2\ [B]}{[A]\ \textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ [B]}$$

While loops

$$\frac{[A \wedge b \wedge \color{red}{t = t_0}]\ \text{p}\ [A \wedge \color{red}{t < t_0}]\ \color{red}{A \wedge b \implies t \geq 0}}{[A]\ \textbf{while}\ b\ \textbf{do}\ p\ \textbf{od}\ [A \wedge \neg b]}$$

Where t is an arithmetic expression like e.g. $x \leq 6$

3.5 Deriving a postcondition - $\{A\}$ x := a $\{?\}$

$\vdash \{A\}$ x := a $\{\exists x'.(A[x/x'] \wedge x = a[x/x'])\}$

Example:

$$\vdash \{x = 2\}\ \text{x} := 3 \left\{ \exists x'. \left(\underbrace{\underbrace{\underbrace{(x = 2)[x/x']}_{\exists! x'(x'=2)} \wedge \underbrace{x = (3[x/x'])}_{x=3}}_{\underbrace{true \text{ and } x = 3}_{x=3}}} \right) \right\}$$

Big paranthesis are not fixed out of lack for motivation.

3.6 Weaker vs. Stronger Assertions

A is <u>weaker</u> than B iff $B \implies A$.
"It is easier to satisfy A, because if I satisfy B I automatically also satisfy A."
Weakest assertion: *true*

A is <u>stronger</u> than B iff $A \implies B$.
"If I satisfy B it does not automatically follow that I satisfy A, but if I do satisfy A, I automatically satisfy B aswell."
Strongest assertion: *false*

3.7 Finding/proving the weakest precondition $wlp(p, B)$

1. Find $wlp(p, B)$
2. Prove $A \implies wlp(p, B)$

Ad 1.
For $p = p_1; ...; p_n$ compute:

1. $wlp(p_n, B)$
2. $wlp(p_{n-1}, wlp(p_n, B))$
3. $wlp(p_{n-2}, wlp(p_{n-1}, wlp(p_n, B))$
   $\vdots$
4. $wlp(p_1, wlp(..., wlp(p_n, B)))$

To compute $wlp(p, B)$ replace variables with their values and use implications, $\wedge$-Operators and similar to make them into first-order logic formulas.

Example:

$$wlp(\textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := -1, z > 0) =$$

$$(x > 0 \implies wlp(z := 1, z > 0)) \wedge (x \leq 0 \implies wlp(z := -1, z > 0)) =$$

$$(x > 0 \implies \underbrace{1 > 0}_{true}) \wedge (x \leq 0 \implies \underbrace{-1 > 0}_{false}) =$$

$$\underbrace{(x > 0 \implies true)}_{true} \wedge \underbrace{(x \leq 0 \implies false)}_{\neg(x \leq 0)} =$$

$$true \wedge \neg(x \leq 0) =$$
$$\neg(x \leq 0) =$$
$$\underline{x > 0}$$

Then use the computed precondition $x > 0$ to compute the next higher precondition for nested $wlp(...)$.

3.8 Weakest Precondition - Rules

3.8.1 Partial Correctness - WLPs

- $wlp(x := a, B) = B[a/x]$
- $wlp(\textbf{skip}, B) = B$

- $wlp(\textbf{abort}, B) = true$
- $wlp(p_1; p_2, B) = wlp(p_1, wlp(p_2, B))$
- $wlp(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, B) = (b \implies wlp(p_1, B) \land \neg b \implies wlp(p_2, B))$
- $wlp(\textbf{while } b \textbf{ do } p \textbf{ od}, B) = I$

3.8.2 Total Correctness - WPs

- $wp(x := a, B) = B[a/x]$
- $wp(\textbf{skip}, B) = B$
- $wp(\textbf{abort}, B) = false$
- $wp(p_1; p_2, B) = wp(p_1, wp(p_2, B))$
- $wp(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, B) = (b \implies wp(p_1, B) \land \neg b \implies wp(p_2, B))$
- $wp(\textbf{while } b \textbf{ do } p \textbf{ od}, B) = I$

3.9 Verification Conditions - VCs

3.9.1 Partial Correctness

- $VC(x := a, B) = true$
- $VC(\textbf{skip}, B) = true$
- $VC(\textbf{abort}, B) = true$
- $VC(p_1; p_2, B) = VC(p_2, B) \land VC(p_1, wlp(p_2, B))$
- $VC(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, B) = VC(p_1, B) \land VC(p_2, B)$
- $VC(\textbf{while } b \textbf{ do } p \textbf{ od}, B) =$

  $(I \land \neg b) \implies B$

  $\bigwedge$

  $(I \land b) \implies wlp(p, I)$

  $\bigwedge$

  $VC(p, I)$

3.9.2 Total Correctness

- $VC(x := a, B) = true$
- $VC(\textbf{skip}, B) = true$
- $VC(\textbf{abort}, B) = true$
- $VC(p_1; p_2, B) = VC(p_2, B) \wedge VC(p_1, wp(p_2, B))$
- $VC(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, B) = VC(p_1, B) \wedge VC(p_2, B)$
- $VC(\textbf{while } b \textbf{ do } p \textbf{ od}, B) =$

  $(I \wedge \neg b) \implies B$

  $\bigwedge$

  $(I \wedge b) \implies t \geq 0$

  $\bigwedge$

  $(I \wedge b \wedge t = t_0) \implies wp(p, I \wedge t < t_0)$

  $\bigwedge$

  $VC(p, I \wedge t < t_0)$

  Choose t like that, that t decreases in the loop and $t \geq 0$

3.10 Prove validity of triple

$\{A\}$ p $\{B\} \implies VC(p, B) \wedge (A \implies wlp(p, B))$

$[A]$ p $[B] \implies VC(p, B) \wedge (A \implies wp(p, B))$

3.11 Finding an Invariant

1. Introduce a loop counter e.g. k
2. Rewrite loop assignments like this:

   If the assignment is $x := x + 5$

   We formulate: $x_{k+1} = x_k + 5$

   Further: $x_k = x_0 + 5 \cdot k$

   Then replace x_0 with the assignment before the loop (e.g. if $x := 0 \rightarrow x_k = 5 \cdot k$)

   See useful formulas below for rewriting sums
3. Reform <u>one</u> equation such that it expresses k

   e.g. $x = 2k \rightarrow k = \frac{x}{2}$

4. Concatenate all other expressions with $\wedge$
5. Add the loop condition b, such that the condition also holds after the loop is finished

   e.g. **while** $z > 0$ **do** ... $\rightarrow z \geq 0$ if z is gradually decreased in the loop
6. Also consider adding a clause from the precondition, if seem fitting

To prove a triple, always start with $A \implies wlp(p, B)$ or $wp(p, B)$!
This way you can cross-check your invariant.

For total correctness we also need a t. Choose/try t such that it has a relation with the loop condition b.
For example if:
  while $z > 0$ **do** ...
We choose $t = z$.

3.11.1 Useful formulas

$x := x + 2 \rightarrow x_k = x_0 + 2k$

$x := x + 2$
$y := y - 4 \cdot x \rightarrow y_k = y_0 - 4 \cdot \frac{x^2 + x}{2}$ (-x0)

Beware of your loop variable k in the sums in the loop bodies! If a variable is used before its new assignment, you have to use $\sum_1^{k-1}$. If the variable has an assignment <u>other than 0</u>, you also have to add its initial value explicitly after the sum (also look if there is a multiplication present, if so you have to multiply it aswell)!

Beware if you have sum of just a variable, which is not the loop variable! Then you cannot just use the gaussian sum!
Example: Let $y = 10 - 2k$
Let $z = 6 - k$

$$
\begin{aligned}
x &= x + 2 \cdot y - 4 \cdot z + 5 \\
x_{k+1} &= x_k 2 \cdot y_k - 4 \cdot z_k + 5 \\
x_k &= x_0 + 2 \cdot \sum_{1}^{k} y - 4 \cdot \sum_{1}^{k-1} z - 4 \cdot z_0 + 5k \\
x_k &= x_0 + 2 \cdot \sum_{1}^{k} (10 - 2k) - 4 \cdot \sum_{1}^{k-1} (6 - k) + 5k \\
x_k &= x_0 + 2 \cdot \sum_{1}^{k} 10 + 2 \cdot \sum_{1}^{k} -2k - 4 \cdot \sum_{1}^{k-1} 6 - 4 \cdot \sum_{1}^{k-1} -k + 5k \\
x_k &= x_0 + 2 \cdot 10k - 4 \cdot \sum_{1}^{k} k - 4 \cdot 6 \cdot (k-1) + 4 \cdot \sum_{1}^{k-1} k + 5k \\
x_k &= x_0 + 20k - 4 \cdot \frac{k^2 + k}{2} - 24 \cdot (k-1) + 4 \cdot \frac{k^2 - k}{2} + 5k
\end{aligned}
$$

4 Model Checking

4.1 Kripke structure

A Kripke structure is denoted as $M = (S, S_0, R, AP, L)$ where:

- S is a (finite) set of states S
- $S_0 \subseteq S$ is the set of initial states
- $R \subseteq S \times S$ is a transition relation such that $\forall s \exists s' : (s, s') \in R$ (= every node has at least one successor)
- AP is some finite set of atomic propositions
- $L : S \to 2^{AP}$ is a function that labels each state with the set of those atomic propositions that are true in that state

A label in a note e.g. $\{p, q\}$ means that p and q are *true* in this state (all other atoms are *false*).

A **path** is denoted as $\pi = s_0, s_1, ...$
A subpath of π is denoted as π^i where i is the start index of the path (e.g. $\pi^1 = \cancel{s_0}, s_1, ...$)

4.1.1 Temporal Logic structures

Temporal logic = "Logic that changes over time. For example <u>now</u> x is *true*, tomorrow it might be *false* and the day after that it might be *true* again."

4.1.2 CTL*

Path Quantifiers

$\mathbf{A}\varphi$	"All paths from given state have property φ"
$\mathbf{E}\varphi$	"At least one path from given state has property φ"

Temporal Operators

$\mathbf{X}\varphi$	"Requires that property φ holds on second state of path (= next state)"
$\mathbf{F}\varphi$	"Assert that property φ will hold at some state on the path (= eventually/in the future)"
$\mathbf{G}\varphi$	"Specifies that property φ holds at every state on the path (= always/globally/everywhere)"
$\varphi\mathbf{U}\psi$	"The property φ is satisfied until some point in the future where ψ holds"

"In the Kripke structure M the state s satisfies φ":

$M, s \models \varphi$

"In the Kripke structure M there is a path π such that ψ holds":
$M, \pi \models \psi$

"There is a path π in M starting at s such that $M, \pi \models \psi$":
$M, s \models \mathbf{E}\psi$

"For every path π in M starting at s it holds $M, \pi \models \psi$":
$M, s \models \mathbf{A}\psi$

Be careful of potential infinite loops, when e.g. dealing with the F- or other operators!

4.1.3 CTL

In CTL path quantifiers and temporal operators **always** occur in pairs.
E.g. only $\mathbf{AX}\varphi$, $\mathbf{EX}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{EF}\varphi$, ...

4.1.4 ACTL* & ACTL

ACTL* is a sublogic from CTL* where only universal path quantification **A** is allowed (no **E** quantifier).

ACTL is a sublogic of CTL and combines both restrictions from CTL and ACTL*.

4.1.5 LTL

The logic LTL only uses path formulas. This means, that we restrict CTL* to disallow path quantification.

4.2 Marking algorithm - CTL Model Checking

Compute all state s of of a Kripke structure M, where $s \models \phi$ for a given formula ϕ.

The "level" or "depth" of a formula is determined by its amount of temporal operators.

1. Determine all subformulas with layers.

   E.g. $\neg\mathbf{EGEF}\neg p \rightarrow$
   $\phi_0 = p, \psi_0 = \neg p$
   $\phi_1 = \mathbf{EF}\neg p, \psi_1 = \mathbf{EF}\neg p$
   $\phi_2 = \mathbf{EGEF}\neg p, \psi_2 = \neg\mathbf{EGEF}\neg p$

2. Subformulas can then be shortened

   E.g. $\mathbf{EG}\underbrace{\mathbf{EF}\neg p}_{q}$ can also be written/considered as $\mathbf{EG}q$

3. Go through the derived subformulas and see if states satisfy the formula (proceed from ϕ_0 to ϕ_n). Mark every state along a path.

   E.g. $s_0\{p\} \rightarrow s_1\{\neg p\}$ and the formula is $\mathbf{EF}\neg p$ we mark both s_0 and s_1

4. Repeat step 3. until there are no more subformulas

5. Delete marked nodes, which do not have a marked successor state with the formula

6. The resulting states with the complete fromula are the states, that satisfy the given formula.

4.3 LTL Model Checking

Check wether $\mathbf{E}\neg\phi$ does not hold (transform **A**s to **E**s by negating).

A node in this algorithm looks like this: (S, i, s_k)
Where S is the set of subformulas, i is an index and s_k is a state from the Kripke structure M.

1. Construct a set S with all consistent subformulas of the formula ϕ to check.

   The set must include the formula itself we want to check and depending on the underlying Kripke structure M some parts of the formula may also be fixed. For the rest we can construct a negated variant aswell.

   E.g. for $s_0 \models \mathbf{EFG}p$ and $s_0\{p\}$ we can have $\mathbf{FG}p, \mathbf{G}p, \neg\mathbf{G}p$ and p in S. This is because $\mathbf{FG}p$ is the formel itself, which we want to check, $\mathbf{G}p$ is not bound (the **F** only indicates, that it must be true some time in the future) so we can also negate it and s_0 satisfies p, so we must include this aswell.

   E.g. $\mathbf{EGF}\neg p$ we would get $\mathbf{GF}\neg p, \mathbf{F}\neg p$ and p for $s_0\{p\}$ because **G** binds $\mathbf{F}\neg p$, because it must be true globaly.

2. Make a transition to another node, if either S or the index gets updated

   The set S is updated, if a subformula is fulfilled and a transition to another state is in M

3. Check if there is a node with index 0 which appears in a cycle

   If this is the case $\mathbf{E} \models \phi$ (the subformulas are satisfied)

   Otherwise $\not\models \mathbf{E}\phi$

4.4 CTL* Model Checking

Combination of the techniques for CLT and LTL model checking.
We use the Marking Algorithm for dealing with state formulas and the LTL algorithm for formulas $\mathbf{E}\phi$.

4.5 (Bi-)Simulation

Two structures are **equivalent**, if they satisfy the same CTL* formulas.

$$M \equiv M' \text{ iff for every formula } \varphi, M \models \varphi \Longleftrightarrow M' \models \phi$$

Two structures are in **preorder** to one another of they preserve ACTL* properties:

$$M' \succeq M \text{ iff for every ACTL* formula } \varphi, M' \models \varphi \implies M \models \varphi$$

If $M_1 \preceq M_2$ then we say M_2 **simulates** M_1

4.5.1 Bisimulation Game

There is a **Spoiler**, which makes moves and a **Duplicator**, which has to duplicate the Spoiler's move in the other structure. The current selection of the states by the players are marked by pebbles.

1. **Spoiler** chooses an initial state of either M or M' (place a pebble there)
2. **Duplicator** has to choose the same initial state on the other structure (node that has the same labelling).
3. **Spoiler** now can move either of the places pebbles to a successor state
4. **Duplicator** now has to answer accordingly, by moving the other pebble to a state wich is labeled equivalently
5. If the **Duplicator** cannot recreate the **Spoiler**'s move, the structures are <u>not bisimilar</u>, else the structures <u>are equivalent</u> (but possible infinite duration)

If we have $M \preceq M'$ and $M \succeq M'$, we still don't have $M \equiv M'$!

4.5.2 Simulation Game

The Simulation Game is similar to the Bisimulation Game, with the only difference being, that the Spoiler cannot choose the pebble to move, rather it always playes with the same pebble (= in the same structure).

If M' simulates M, then we write $M \preceq M'$.

1. **Spoiler** chooses an initial state of $\mathbf{M}'$ (place a pebble there)
2. **Duplicator** has to choose the same initial state on the other structure M (node that has the same labelling).
3. **Spoiler** now moves to a successor state
4. **Duplicator** has to answer accordingly, by moving its pebble to a state wich is labeled equivalently
5. If the **Duplicator** cannot recreate the **Spoiler**'s move, M' $\underline{\text{does not simulate}}$ M

4.6 Abstraction

A structure M is abstracted, when we want to reduce/limit the amount of states we have in a system.
According to some equivalence relation we combine multiple states into an abstract state. These reduced states inherit some properties from their initial state such as:

- The start/initial state
- Transitions to other (now abstract) states
- Transitions to other state, which are now contained in the same abstract state become a self loop of the abstract state

For construction of the abstract states we can use predicates to define which states get grouped together (e.g. a predicate might be $p_1(s) = (s.x > s.y)$).
One can then check properties (e.g. $\mathbf{G}p_1$) over a abstracted structure, but note that these properties can be "spurious" meaning they are false in the abstract structure, but might not be false in the original unabstracted structure, because for example one state might not be reachable in the original structure. In this case the property might be false for the abstract state, but there is no concrete counterexample, because in the initial structure the property holds.

One can then check properties over the abstracted structure, but note that

4.7 Symbolic encoding of Kripke structures

In our case specified to LTL model checking of reachability (can I reach a node such that, ...) and safety (does it hold that, ...).

We translate a Kripke structure from its explicit encoding (a graph) to a formula:

1. Notate the formula which holds for the starting state
2. The notated formula implies " $\implies$ " it's neighbouring formulas, which are connected to the current state (disjuncted - "$\vee$")

   The implied formulas are denoted with a prime (e.g. $a \implies (\neg a' \wedge b')$)

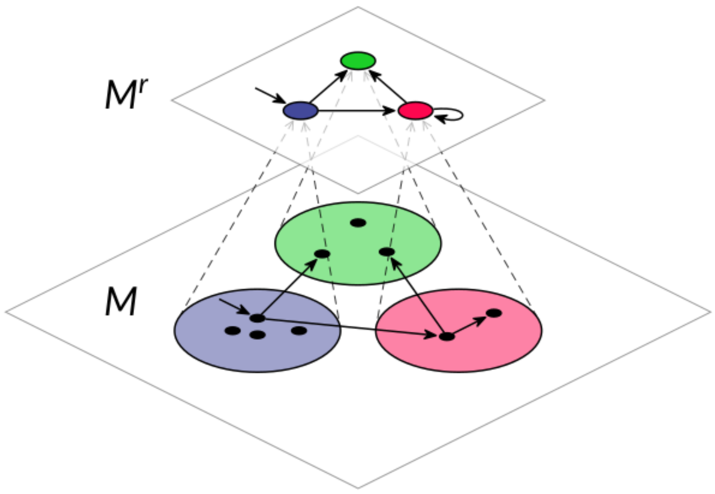


Figure 4: Reduced model M^r obtained by existential abstraction from M

3. Conjunct ("$\wedge$") the next formula, which is one of the connected nodes
4. Repeat step 2. and 3. until all nodes are processed

5 Solved Exercises

5.1 Many-one reduction from HALTING to REACHABLE-CODE - Exercise sheet 2023

Let (Π, I) be an arbitrary instance of HALTING. We construct the REACHABLE-CODE instance (Π', n) as follows:

```
Π'(String S) {
    Π(I)
    return 1;
}
```

Now we set the line number n from REACHABLE-CODE to the last line in Π' (the `return 1;` statement).

Show HALTING $(\Pi, I) \Leftrightarrow$ REACHABLE-CODE (Π', n).

1. HALTING $(\Pi, I) \Rightarrow$ REACHABLE-CODE (Π', n)

   If the call to $\Pi(I)$ halts, the last line of Π' is executed. This marks a positive instance for HALTING and REACHABLE-CODE.

2. HALTING $(\Pi, I) \Leftarrow$ REACHABLE-CODE (Π', n)

   If the last line of Π' is executed and Π' returns, the call to $\Pi(I)$ must also have terminated, which marks a positive instance for HALTING and REACHABLE-CODE.

5.2 Prove correctness of many-one reduction from HALTING to HALTING-X - Exam 26.02.2021

HALTING-X
INSTANCE: Two programs Π_1, Π_2 that take a string as input
QUESTION: Does there exist at least one input string I such that both Π_1 and Π_2 halt on I?

Provided reduction function $f(\Pi, I) = (\Pi_1, \Pi_2)$:

```
Π₁(string S) = call Π(S); return;
Π₂(string S) = if (S ≠ I {while(true){}}; return;
```

Solution (try):
"$\Rightarrow$" direction: Assume that Π halts on input I (positive instance of HALTING). Then also Π_1 halts, since it calls Π, which by our assumption halts. Since our input is I in Π_2, S is equal to I, which makes the `if` condition false and therefore Π_2 halts too. Since Π_1 and Π_2 both halt, it is also a positive instance of HALTING-X.

"$\Leftarrow$" direction: Since for a positive instance of HALTING-X both Π_1 and Π_2 halt on I, also Π has to halt on I. If Π_1 halts, Π also halts, since for Π_1 to finish executing Π cannot run infinitely. Since the input for Π_2 is I the `if` condition will always be false and Π_2 will halt for input I.

5.3 Prove correctness of many-one reduction from HALTING to BITFLIP-HALTING - Exam 20.06.2023

BITFLIP-HALTING
INSTANCE: A program Π that takes a string as input, and a string I.
QUESTION: Does Π halt on I, but not halt on the bit-flipped string $flip(I)$?

Provided reduction function $g((\Pi, I)) = (\Pi_1, I_1)$, where $I_1 = flip(I)$:

```
$\Pi_1$(string $S$) {
    if ($S$ = $flip(I)$) {
        return;
    }
    $\Pi(S)$;
    return;
}
```

Solution (try):
Positive instance of Co-HALTING: program Π does not halt on I.
Positive instance of BITFLIP-HALTING: program Π halts on I, but not on the bit-flipped string $flip(I)$.

"$\Rightarrow$" direction: Assume an arbitrary positive instance of Co-Halting such that Π does not halt on I.

Π_1 halts: Since $I_1 = flip(I)$, Π_1 halts on input I, because the `if` condition ($S = flip(I)$ or in Π_1: $I_1 = flip(I)$) is satisfied.

Π_1 does not halt: If the input to Π_1 is $flip(I)$ the input $I_1 = flip(flip(I))$ is equal to I. Since $I_1 \neq flip(I)$, the `if` condition is not satisfied and $\Pi(S)$ ($\Pi(I_1) = P(I)$, since $I_1 = I$) is executed. Since by assumption $\Pi(I)$ does not halt, Π_1 also does not halt.

"$\Leftarrow$" direction: Assume that Π_1 does halt on I and does not on $flip(I)$.

Since Π_1 does not halt on $flip(I)$, the `if` condition must be false. This is the case, since $I_1 = flip(I)$ and $I = flip(I)$, so $I_1 = flip(flip(I)) = I$. This means that $\Pi(I_1)$ ($= \Pi(I)$) is executed. Since Π_1 does not halt by assumption, $\Pi(I)$ also cannot halt.

(Because Π_1 does halt on I, the `if` condition must be satisfied. Since the function g maps the input I to $I_1 = flip(I)$, the `if` condition $S = flip(I)$ is satisfied and Π_1 halts without executing $\Pi(I)$.)

5.4 Prove correctness of many-one reduction from DOMINATING SET to DS-TRIANGLE - Exam 19.05.2023

DS-Triangle
INSTANCE: A *triangle-graph* $G = (V, E)$, and an integer k.
QUESTION: Does there exist a dominating set of vertices of size at most k, i.e., is there a set $S \subseteq V$ with $|S| \leq k$ such that for every vertex $v \in V$ it either holds $v \in S$ or there is some $w \in S$ such that $(v, w) \in E$.

Recall that the (standard) DOMINATING SET problem is defined over *arbitrary* undirected graphs (together with an integer k) and has the same question.

Provided reduction function g with

$$g((G, k)) = (f(G), k + 1).$$

Further $f(G) = (V', E')$ is defined for a graph $G = (V, E)$, let $\{a, b, c\}$ be a set of fresh vertices. Moreover we define:

$$V' = V \cup \{a, b, c\},$$
$$E' = E \cup \{(a, b), (b, c), (c, a)\}.$$

Show that G, k is a positive instance of DOMINATING SET iff $g((G, k))$ is a positive instance of DS-TRIANGLE.

Solution (try):
DS $\Rightarrow$ DS-T: Let there be an $S \subseteq V$ with $|S| \leq k$, s.t. S is a positive instance of an arbitrary graph G of the DOMINATING SET problem. If we now apply the function g to our instance, according to $f(G)$ we add three new vertices to our graph $V' = V \cup \{a, b, c\}$, which are internally connected in a triangle $E' = E \cup \{(a, b), (b, c), (c, a)\}$. The bound k is also increased by one $(k + 1)$. Since at least one of the new vertices has to be in S we select one of the vertices, no matter if there is a path from V to any new vertice a, b or c. Since we select one of the new vertice to be in S, the new graph also forms a dominating set, because S already was a positive instance.

DS $\Leftarrow$ DS-T: Assume we have a positive instance of DS-TRIANGLE with (S, k). Since S forms a dominating set within bound k in a triangle-graph by assumption, this instance also forms a positive intance of DOMINATING SET for an arbitrary graph.