Symbolic Methods

The finite-state case

— Part I —

Martin Fränzle

Carl von Ossietzky Universität

FK II, Dpt. Informatik

Abt. Hybride Systeme

What you’ll learn

How to use and manipulate predicative descriptions of state sets for

- manipulating extremely large finite state spaces (this and the next lecture)
- encoding uncountably infinite state sets (later lectures).

The bottleneck of explicit-state model checking

State explosion

State explosion

- Parallel components are flattened out into a flat state graph via product construction.
- 10 components with 8 states each yield $8^{10} > 10^9$ nodes in transition graph.
- Explicit representation of transition graph with 10^9 *nodes* requires in the order of 10 *GByte* memory.

$\Rightarrow$ Need more compact representation.

1. **On-the-fly methods.**
2. **Partial-order reduction:** Enforces an ordering for causally independent events.
3. **Symbolic model checking:** Uses predicates for representing sets of vertices and sets of edges.

Symbolic model checking

Idea: *Predicative representation of node sets and edge sets*, e.g.
$\mathsf{In} \Rightarrow \mathsf{Closed}$ instead of

$$\left\{\begin{array}{c}(\mathsf{Empty},\mathsf{Open}),(\mathsf{Empty},\mathsf{Closing}),(\mathsf{Empty},\mathsf{Closed}),(\mathsf{Empty},\mathsf{Opening}),\\ (\mathsf{Appr.},\mathsf{Open}),(\mathsf{Appr.},\mathsf{Closing}),(\mathsf{Appr.},\mathsf{Closed}),(\mathsf{Appr.},\mathsf{Opening}),(\mathsf{In},\mathsf{Closed})\end{array}\right\}$$

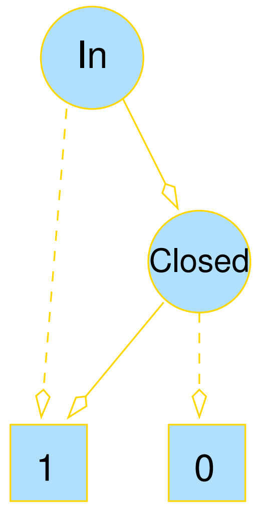
Problem: There are various equivalent predicates, some of which are all but compact:

$$\begin{array}{c}(\mathsf{Empty} \wedge \mathsf{Open}) \vee (\mathsf{Empty} \wedge \mathsf{Closing}) \vee (\mathsf{Empty} \wedge \mathsf{Closed}) \vee (\mathsf{Empty} \wedge \mathsf{Opening}) \vee\\ (\mathsf{Appr.} \wedge \mathsf{Open}) \vee (\mathsf{Appr.} \wedge \mathsf{Closing}) \vee (sfAppr. \wedge \mathsf{Closed}) \vee \ldots \vee (\mathsf{In} \wedge \mathsf{Closed})\end{array}$$

Symbolic model checking

BDD-based:

- Based on a data structure that assigns the same (compact) representation to equivalent predicates.



- Normalizes representation while performing the graph traversal.

SAT-based:

- Based on efficient heuristics for checking satisfiability of large propositional formulae.

$$\text{In} \Rightarrow \text{Closed}$$

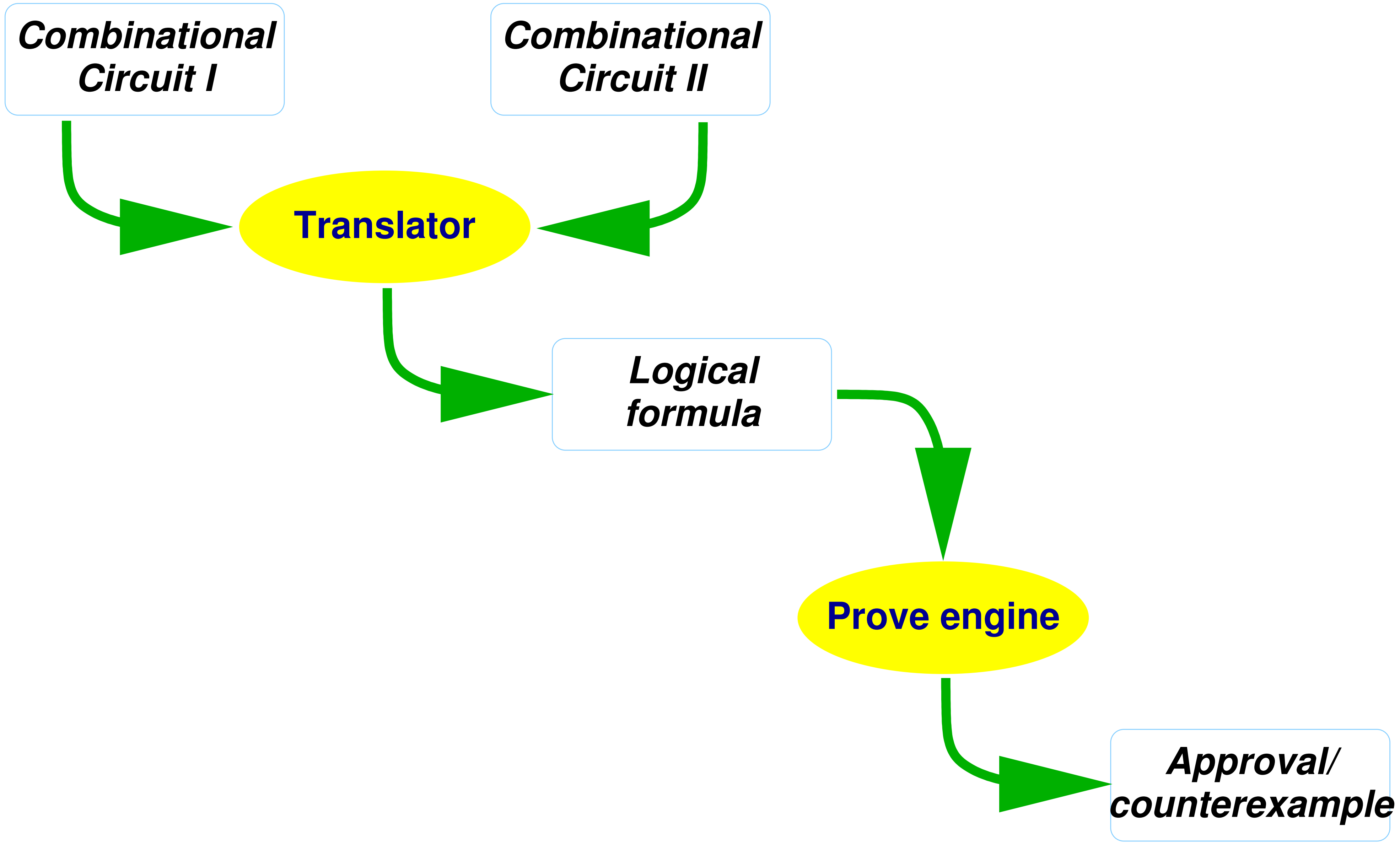
$$(\text{Empty} \wedge \text{Open}) \vee (\text{Empty} \wedge \text{Closing}) \vee \ldots$$

- No semantic analysis upon graph traversal.

Symbolic techniques I:

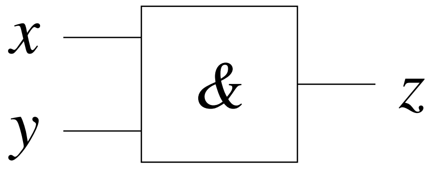
Equivalence check for combinational circuits

The general framework

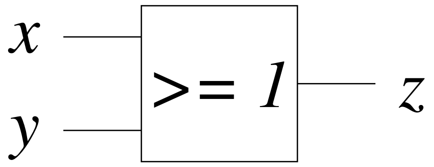


Mapping circuits to formulae

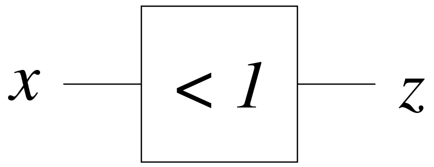
A gate is mapped to a propositional formula formalizing its invariant:



$\mapsto \quad x \land y \Leftrightarrow z$



$\mapsto \quad x \lor y \Leftrightarrow z$



$\mapsto \quad \neg x \Leftrightarrow z$

$\vdots \quad \mapsto$ combinations thereof.

Circuit behavior corresponds to conjunction of all its gate formulae.

Formalizing circuit equivalence

- Given two circuits C and D, we obtain formulae ϕ_C and ϕ_D,
- furthermore, have correspondence lists $I \subset \mathit{Node}_C \times \mathit{Node}_D$ and $O \subset \mathit{Node}_C \times \mathit{Node}_D$ for in- and outputs.
- generate formula $\mathit{Eq}(C, D) =$

$$\left(\phi_C \wedge \phi_D \wedge \bigwedge_{(i,j)\in I} (i \Leftrightarrow j)\right) \Longrightarrow \bigwedge_{(o,p)\in O} (o \Leftrightarrow p)$$

☺ $\neg\mathit{Eq}(C, D)$ is satisfiable if and only if circuits are not equivalent.

☺ Any satisfying assignment yields a counterexample.

Efficient search for satisfying valuations

Enumerating valuations

… is completely out-of-scope:

- When comparing two circuits of (only) 10.000 nodes, we need to explore $4 \cdot 10^{6020}$ possible valuations.
- If we were able to explore $10^8 \frac{\text{valuations}}{s}$, this would take $7 \cdot 10^{6017}$ years.

Enumerating only inputs is *considerably* more efficient, but still out-of-scope:

- When comparing two circuits with 100 input nodes, we need to explore $1.3 \cdot 10^{30}$ possible valuations.
- If we were able to explore $10^8 \frac{\text{input valuations}}{s}$, this would still take $9.6 \cdot 10^{15}$ years.

Alternative approach

1. Employ an efficient translation to a structurally simple subset of propositional logic (e.g., CNF)

What does “efficient translation” mean?

2. Use fast algorithms for detecting satisfiability of such propositional formulae

Ingredients:

- inference mechanisms (to save searching all valuations)
- heuristics (due to NP-completeness of the problem)

Translation to CNF: naive approach

- Translation to CNF can be accomplished by iterated application of the distributive laws (plus de Morgan etc.):

$$\begin{aligned}
(\phi \wedge \psi) \vee \chi &\equiv (\phi \vee \chi) \wedge (\psi \vee \chi) \\
\neg(\phi \wedge \psi) &\equiv (\neg\phi) \vee (\neg\psi) \\
\neg(\phi \vee \psi) &\equiv (\neg\phi) \wedge (\neg\psi) \\
\phi \implies \psi &\equiv (\neg\phi) \vee \psi \\
\neg\neg\phi &\equiv \phi
\end{aligned}$$

  The latter 4 rules allow to transfer the propositional formula into an and-or-tree in negation normal form (where negations do only occur in front of atomic propositions), first rule allows to reshuffle $\wedge$ and $\vee$ until in CNF.

- This tends to provoke an *exponential blowup* of the formula, as the distributive law duplicates subformulae.

- At least in the worst case, this is unavoidable for any *semantics-preserving* transformation of propositional formulae (unless P=NP):
  Checking validity of arbitrarily structured propositional formulae is in NP, while validity of CNF is linear.

Translation to CNF

? Do we really need *preservation of semantics* in the translation step?

! Ultimately, we are only interested in satisfiability, thus *satisfiability-preserving* translation should be good enough...

Idea: Design a translation that

1. translates arbitrary propositional formulae to *equi-satisfiable* CNFs
   (Validity etc. need not be preserved!)
2. permits simple retrieval of a satisfying valuation of the original formula from any satisfying valuation of the CNF
   (Otherwise, counterexample generation wouldn't work!)

Tseitin Transformation [Tseitin 1968]

Idea: Introduce a fresh propositional variable for each subformula in order to represent its truth value;
axiomatize the relation between satisfaction of a (sub-)formula and satisfaction of its immediate subformulae via CNF clauses.

Alg.: Given a propositional formula ϕ generate a CNF η as follows:

1. For each subformula ψ of ϕ take a fresh propositional variable $[\psi]$,
2. translate the topmost operator of each subformula ψ by

$$\begin{array}{rcrl}
\psi = \psi_1 \vee \psi_2 & \rightsquigarrow & & (\neg[\psi] \vee [\psi_1] \vee [\psi]_2) \\
 & & \wedge & ([\psi] \vee \neg[\psi_1]) \\
 & & \wedge & ([\psi] \vee \neg[\psi_2]) \\
\psi = \psi_1 \wedge \psi_2 & \rightsquigarrow & & (\neg[\psi] \vee [\psi_1]) \\
 & & \wedge & (\neg[\psi] \vee [\psi_2]) \\
 & & \wedge & ([\psi] \vee \neg[\psi_1] \vee \neg[\psi]_2) \\
\psi = \neg\psi_1 & \rightsquigarrow & \multicolumn{2}{l}{(\neg[\psi] \vee \neg[\psi_1]) \wedge ([\psi] \vee [\psi_1])} \\
\psi = v \in V & \rightsquigarrow & \multicolumn{2}{l}{(\neg[\psi] \vee v) \wedge ([\psi] \vee \neg v)}
\end{array}$$

3. collect all the CNF fragments thus obtained; conjoin them by $\wedge$,
4. add (i.e., conjoin it using $\wedge$) the “goal” $[\phi]$.

Tseitin Transformation: Properties

Let ϕ be a propositional formula and ψ the formula obtained from it through Tseitin transformation.

Thm: ϕ and ψ are equi-satisfiable.

Prf: Follows immediately from the following lemmata.

Lemma: If $\sigma : V \to \mathbb{B}$ is a satisfying valuation for ψ then σ is a satisfying valuation for ϕ, and so is any other valuation σ' that coincides with σ on $\mathrm{free}(\phi)$ (i.e. $\sigma'(v) = \sigma(v)$ for each $v \in \mathrm{free}(\phi)$).

Lemma: If $\sigma : V \to \mathbb{B}$ is a satisfying valuation for ϕ then there is a valuation σ' which satisfies ψ and which coincides with σ on $\mathrm{free}(\phi)$.

...and ψ can be obtained from ϕ in linear time!

Tseitin Transformation: Optimizations

There are various optimizations which make the formula obtained even smaller:

1. Common subexpression elimination:
   If a subformula occurs more than once, the CNF clauses for all but the first occurrence can be saved through reuse of the Tseitin variable.

Tseitin Transformation: Optimizations

2. Polarity optimization: If a subformula occurs in positive (negative) context then we can drop the left-to-right (right-to-left) implication in the "definition" of its Tseitin variable without enhancing satisfiability, i.e. without impeding equi-satisfiability:
   - in positive context

$$\begin{aligned} \psi = \psi_1 \vee \psi_2 &\quad\rightsquigarrow\quad (\neg[\psi] \vee [\psi_1] \vee [\psi]_2) \\ \psi = \psi_1 \wedge \psi_2 &\quad\rightsquigarrow\quad (\neg[\psi] \vee [\psi_1]) \wedge (\neg[\psi] \vee [\psi_2]) \\ \psi = \neg\psi_1 &\quad\rightsquigarrow\quad (\neg[\psi] \vee \neg[\psi_1]) \end{aligned}$$

   - in negative context

$$\begin{aligned} \psi = \psi_1 \vee \psi_2 &\quad\rightsquigarrow\quad ([\psi] \vee \neg[\psi_1]) \wedge ([\psi] \vee \neg[\psi_2]) \\ \psi = \psi_1 \wedge \psi_2 &\quad\rightsquigarrow\quad ([\psi] \vee \neg[\psi_1] \vee \neg[\psi]_2) \\ \psi = \neg\psi_1 &\quad\rightsquigarrow\quad ([\psi] \vee [\psi_1]) \end{aligned}$$

Efficient satisfiability check for CNFs

Davis-Putnam-Loveland-Logemann algorithm

Given a CNF $(x \vee \overline{y} \vee \overline{z}) \wedge (x \vee y) \wedge (z) \wedge \ldots)$,

1. **Perform unit propagation:**

   Search for unit clauses (l) or $(\bar{l})$.
   If present then

   - infer appropriate truth value for l (e.g., $l =$ true),
   - propagate it into all clauses:

$$\begin{array}{lcl} (\bar{l} \vee \overline{y} \vee z) & \rightsquigarrow & (\overline{y} \vee z) \\ (l \vee \overline{y} \vee z) & \rightsquigarrow & \varepsilon \end{array}$$

   Repeat this until no further unit literals are present.

$$\begin{array}{c} (x \vee \overline{y} \vee \overline{z}\,) \wedge (x \vee y) \qquad \wedge (z) \qquad \wedge \ldots \\ \downarrow [\text{propagate } z := \text{true}] \\ (x \vee \overline{y} \vee \overline{z}\,) \wedge (x \vee y) \qquad \wedge (z) \qquad \wedge \ldots \end{array}$$

Davis-Putnam-Loveland-Logemann algorithm

Given a CNF $(x \vee \overline{y} \vee \overline{z}) \wedge (x \vee y) \wedge (z) \wedge \ldots)$,

2. **Perform choice:**

   Select an unassigned propositional variable l occurring in the formula.
   If there is one then

   - freely choose a truth value for l (e.g., $l =$ false),
   - propagate it into all clauses,
   - go back to unit propagation (there might be new units now).

   If no variables are left in the formula then a satisfying assignment has been found; if only assigned var.s are left then go to next phase.

$$
\begin{array}{c}
(x \vee \overline{y}) \wedge (x \vee y \qquad) \wedge \ldots \\
\downarrow \text{[choose } x := \text{false]} \\
(\qquad \overline{y}) \wedge (\qquad y \qquad) \wedge \ldots \\
\downarrow \text{[propagate } y := \text{false]} \\
(\qquad y \qquad) \wedge \ldots
\end{array}
$$

Davis-Putnam-Loveland-Logemann algorithm

Given a CNF $(x \vee \overline{y} \vee \overline{z}) \wedge (x \vee y) \wedge (z) \wedge \ldots$,

3. **Upon conflict, perform backtracking:**
   - Find the most recently assigned (by choice) propositional variable l that has a yet unexplored truth-value,
   - unassign all variables that have been assigned after l,
   - complement l's assignment,
   - go back to unit propagation (there might be new units now).

   If no variable with an unexplored truth value is found then the formula is unsatisfiable.

$$\downarrow [\text{unassign } y, x]$$

$$(x \vee \overline{y}) \wedge (x \vee y \quad) \wedge \ldots$$

$$\downarrow [\text{assign } x := \text{true}]$$

$$\ldots$$

DPLL: Main ingredients

1. Manipulates *partial* truth value assignments.
2. checks consistency of partial assignment with clause set,
3. each individual clause can be
   - satisfied (doesn't impose constraints on further completion of the assignment)
   - unsatisfied (proves inconsistency of the partial assignment, and thus of all possible completions)
   - unresolved (to be re-explored upon extension of the assignment)
   - propagating (determines certain extensions of partial asgn.)
4. if partial assignment is inconsistent then *all* extensions are refuted,
5. if consistent then extended through
   - Boolean constraint propagation (in DPLL: unit resolution)
   - choice.

Why DPLL works in practice

- In practice, only a small fraction of the possible assignments is tested, as unit propagation often derives a contradiction from a partial assignment.
- DPLL can (and is) extended by conflict analysis and learning, which derives a minimal reason for a conflict and learns this s.t. the same conflict is never again encountered — despite backtracking.
- Practical procedures tackle instances with $\gg 100.000$ propositions.

DPLL: Algorithmic enhancements

Backjumping: Strictly chronological backtracking, i.e.

- unassign the most recently assigned (by choice) propositional variable $\mathtt{l}$ that has a yet unexplored truth-value, plus all younger assignments,
- assign the complement value to $\mathtt{l}$,

does

☺ safely avoid revisits to an already explored truth assignment,

☹ often flips variables that weren't involved into the actual conflict.

The solution is

1. to maintain an *implication graph* recording the reasons for assignments,
2. to traverse this graph upon conflicts, thereby revealing the reasons for the conflict,
3. do *backjumping* upon conflicts:
   - find the most recently assigned (by choice) propositional variable $\mathtt{l}$ that *is a reason for the conflict* and has a yet unexplored truth-value,
   - unassign all variables that have been assigned after $\mathtt{l}$,
   - complement $\mathtt{l}$'s value.

DPLL: Algorithmic enhancements

Conflict-driven learning: The basic DPLL scheme tends to run into the same inconsistency again and again:

- only part of the chosen variables may contribute to the conflict,
- when backtracking unassigns all of these (due to a reassignment to an older choice variable) then all recorded information about the conflict gets lost,

$\Rightarrow$ same conflict may be rebuilt in the new branch of the backtracking tree,

☹ the overall search tree can contain multiple copies of more-or-less isomorphic failing subtrees.

The solution is

1. to cut the implication graph between the conflict and those choices which are reasons,
2. to collect the assignments immediately left to the cut (the conjunction of these assignments is a reason of the conflict)
3. to add the disjunction of the complements of these assignments to the clause database (this disjunction is called a “conflict clause”)

Example

$A + B$

$A + c + D$

$A + d + E$

$c + F + G$

$d + F + g$

$f + G$

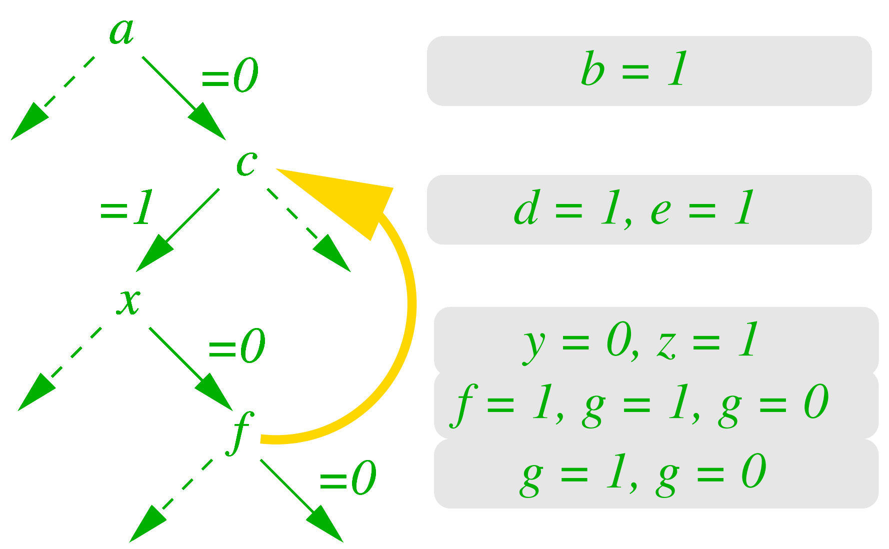
$f + g$

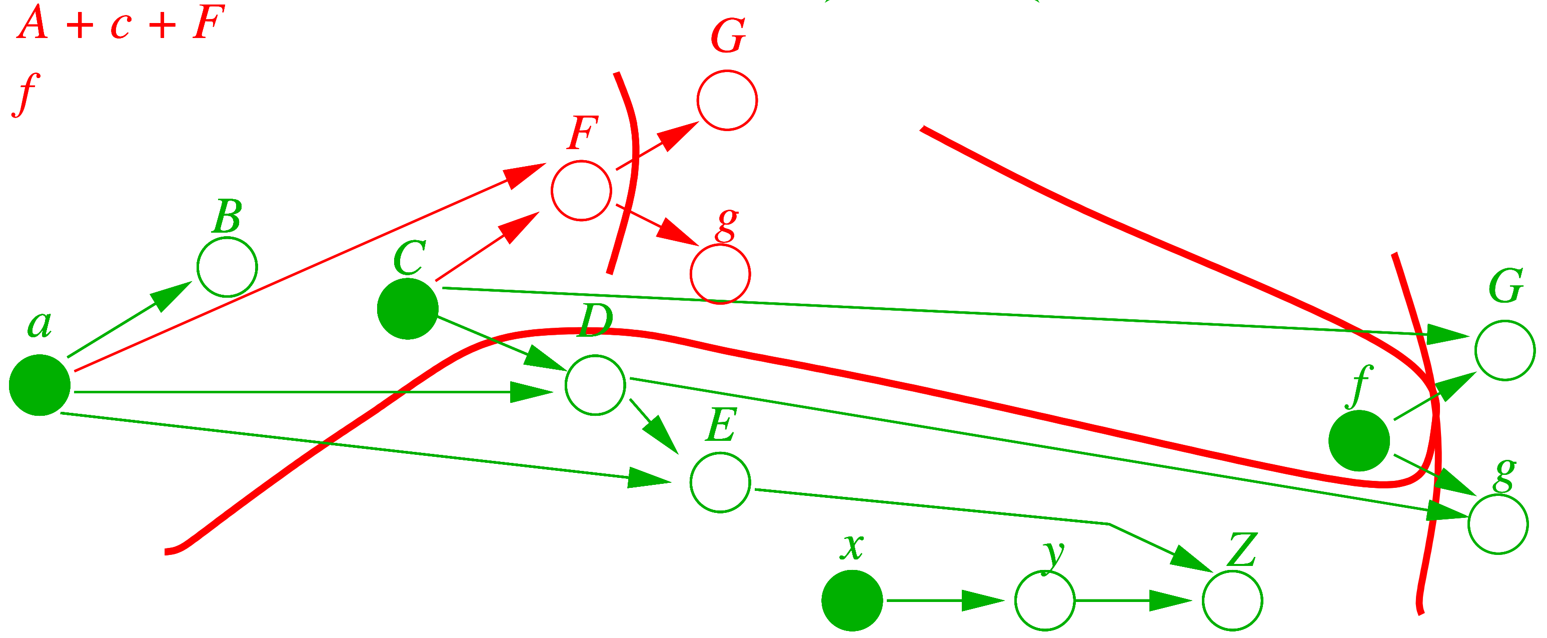
$X + y$

$e + Y + Z$

$A + c + F$

f





DPLL: Algorithmic enhancements

Non-chronological backtracking: If learning is employed then there is no need to explictly reassign choice variables unbound during backtracking:

- conflict clauses learned prevent us from reconstructing an already visited assignment (they become propagating before completing such an assignment),

$\Rightarrow$ after unassignment, we can simply start in phase 1 of DPLL again: backtracking rule is modified to

  - diagnose and learn conflict,
  - find the most recently assigned (by choice) propositional variable l that is a reason and has a yet unexplored truth-value,
  - unassign all variables that have been assigned after l,
  - *unassign* (instead of complement) l's assignment,
  - go back to unit propagation (there might be new units now).

If no variable with an unexplored truth value is found then the formula is unsatisfiable.

DPLL: Algorithmic enhancements

Random restart: The longer a SAT solver runs, the longer and more specific get conflicts detected. A SAT solver getting stuck in the “wrong” part of the search tree thus learns unimportant conflicts. Random restart avoids this by

- interrupting the first few runs of the SAT solver after a while (preferably before it starts learning long conflicts)
- restarting it from scratch, yet preserving the conflicts learned

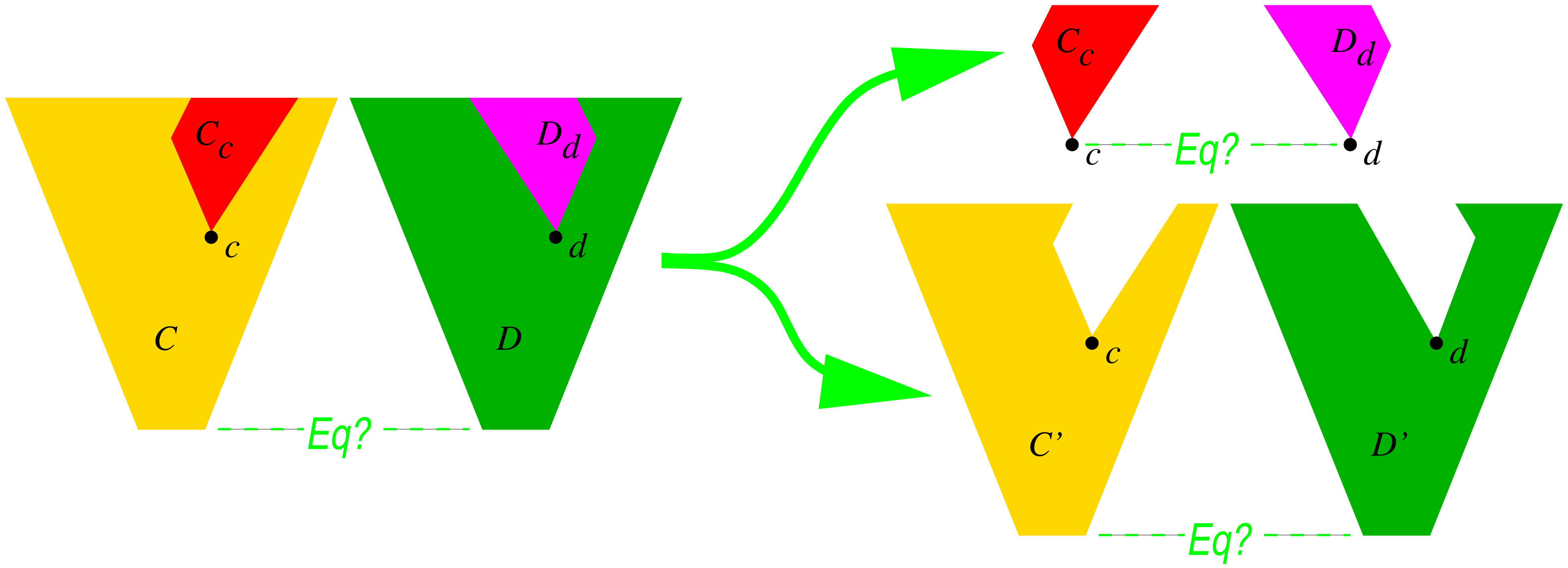
Efficient manipulation of clause database: A lot of optimizations accelerate access to the clause database by appropriate data structures. Particularly effective is to avoid unnecessary visits to clauses (e.g. to any clause whose satisfaction and propagation properties didn’t change since the last visit: watched literal scheme of CHAFF [Moskewicz e.a. 2001]).

Non-monolithic proofs

Automatic discovery of lemmas

for divide-and-conquer approaches

Divide-and-conquer



Given *conjectured* equivalence between nodes c and d of circuits C and D,

1. use equivalence checking to prove or disprove *Eq*$(\mathrm{C}_\mathrm{c}, \mathrm{D}_\mathrm{d})$,
2. if *Eq*$(\mathrm{C}_\mathrm{c}, \mathrm{D}_\mathrm{d})$ holds then C_c and D_d are cut out, thereby adding nodes c and d as a new pair of corresponding inputs
3. check *Eq*$(\mathrm{C}', \mathrm{D}')$ for the remaining circuits.

Eq$(\mathrm{C}_\mathrm{c}, \mathrm{D}_\mathrm{d})$ and *Eq*$(\mathrm{C}', \mathrm{D}')$ can be considerably cheaper than *Eq*(C, D).

Obtaining conjectured equivalences

Randomized divide-and-conquer:

Finds conjectures for node equivalences by a randomized procedure:

1. Randomly generate input vectors and apply them to circuits C and D;
2. find nodes $\mathrm{c} \in \mathit{Node}_{\mathrm{C}}$ and $\mathrm{d} \in \mathit{Node}_{\mathrm{d}}$ which had the same value under all vectors.

Structure-based divide-and-conquer:

Searches for structural matches in the circuits:

1. Candidate nodes c and d depend on the same inputs,
2. building blocks of C_{c} and D_{d} can be matched.

These heuristics search for possible cuts in the circuits. There are similar divide-and-conquer techniques based on cutting the proof tree.

State of the art

- Using divide-and-conquer, multiform decision procedures can be combined.
- Circuits with millions of gates have been tackled!
  [cf. Becker, Fujita, Meinel, Somenzi; Dagstuhl-Seminar 297, 2001.]
- Hardware vendors do directly promote development of such tools:
  - IBM, Intel, Cadence, etc., all develop solver technology in-house
  - Most vendors also market them as pre-packaged EDA components
  - Their research labs are extremely active in the scientific community, ensuring absence of a synchrony gap (in any direction) between academia and industry
- With an interest in software analysis, Microsoft Research also is a major driving force in SAT solving and constraint solving