SAT in Formal Hardware Verification

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Invited Talk SAT'05

St. Andrews, Scotland
20. June 2005

Overview

- Hardware Verification Problems
  - Model Checking
  - Equivalence Checking
- Circuit vs. SAT Simplification Techniques
  - redundancy removal with D-algorithm vs. variable instantiation
- QBF for Verification

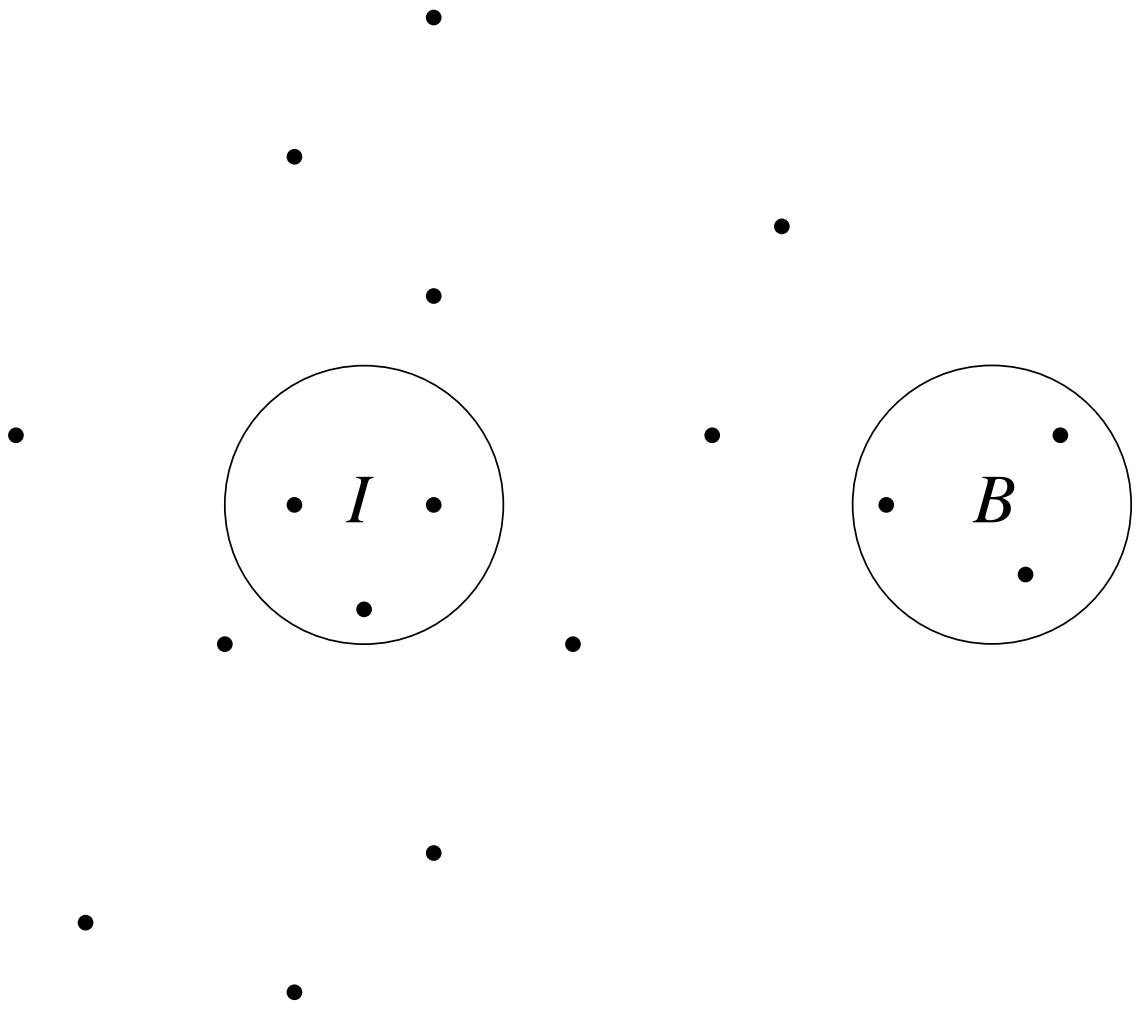
Model Checking

- explicit model checking [ClarkeEmerson'82], [Holzmann'91]
  - program presented symbolically (no transition matrix)
  - traversed state space represented explicitly
  - e.g. reached states are explicitly saved bit for bit in hash table

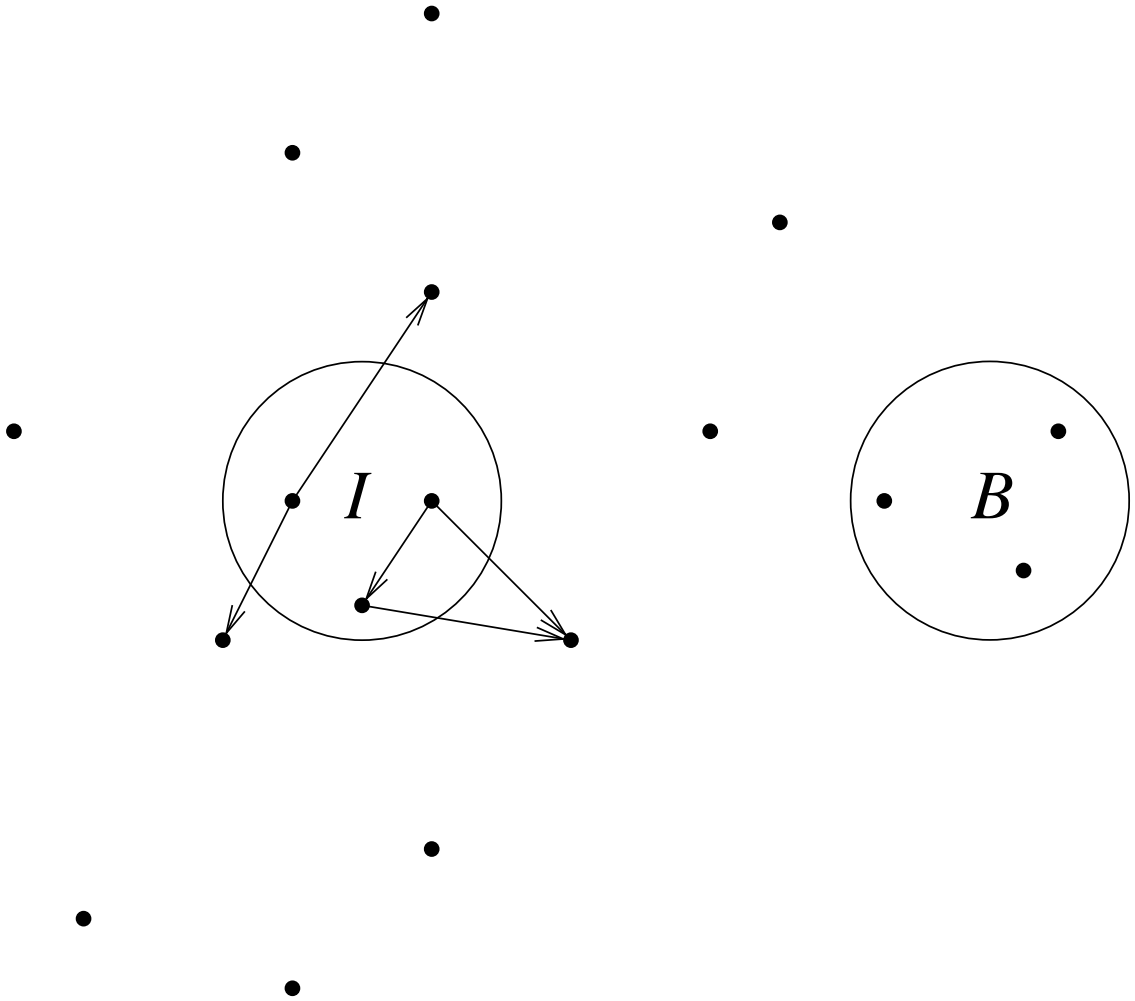
  $\Rightarrow$ State Explosion Problem (state space exponential in program size)

- symbolic model checking [McMillan Thesis'93], [CoudertMadre'89]
  - use symbolic representations for sets of states
  - originally with Binary Decision Diagrams [Bryant'86]
  - Bounded Model Checking using SAT [BiereCimattiClarkeZhu'99]

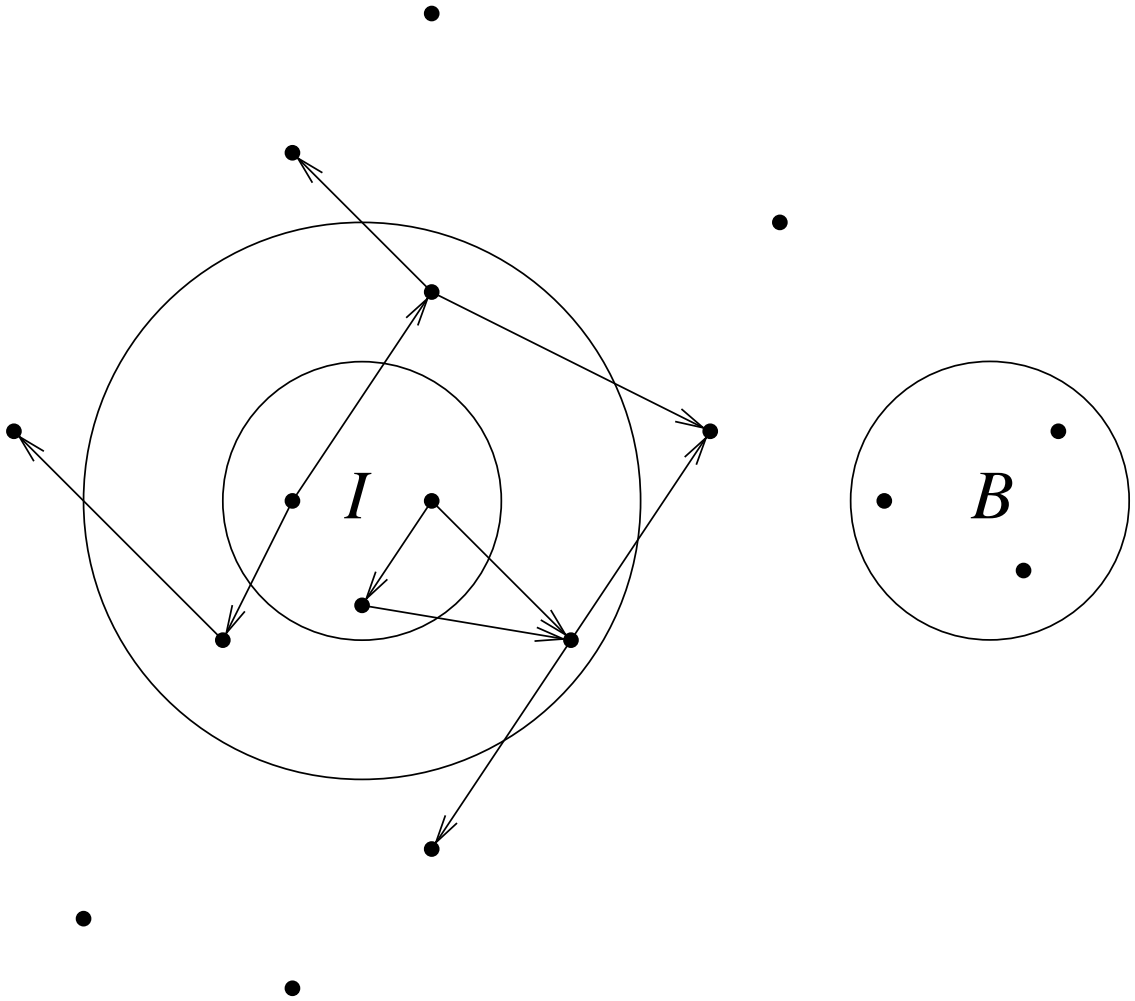
Forward Fixpoint Algorithm: Initial and Bad States



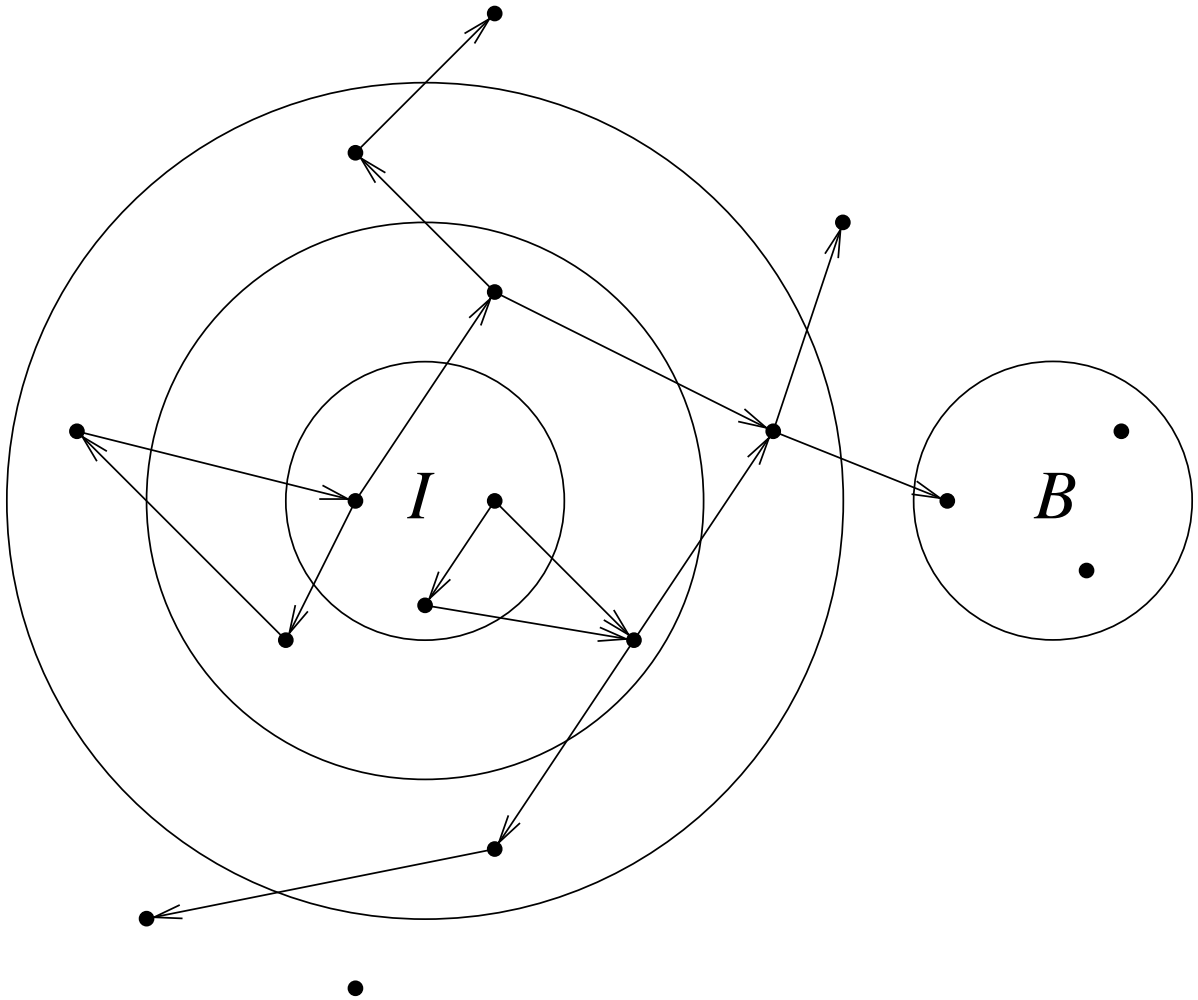
Forward Fixpoint Algorithm: Step 1



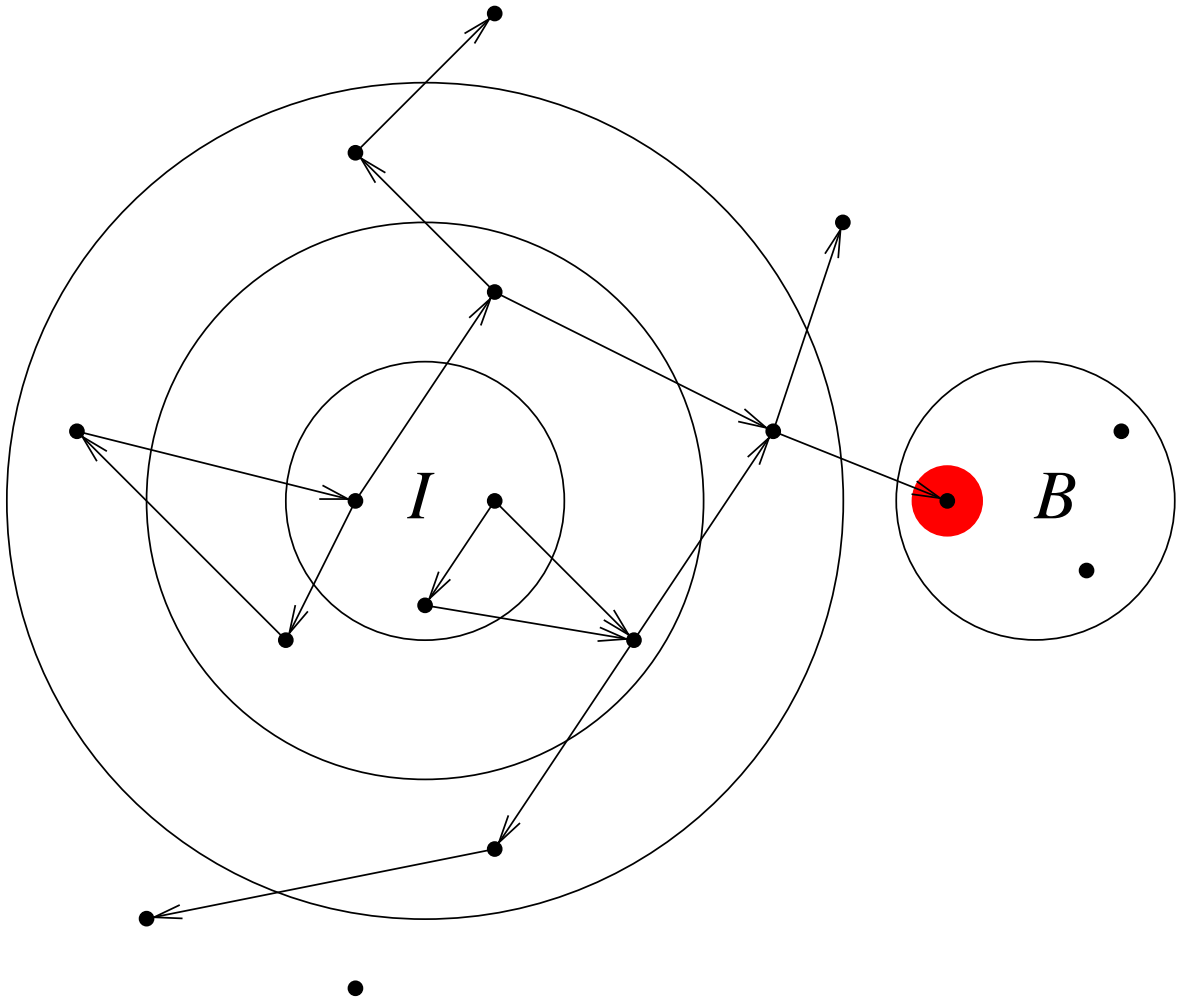
Forward Fixpoint Algorithm: Step 2



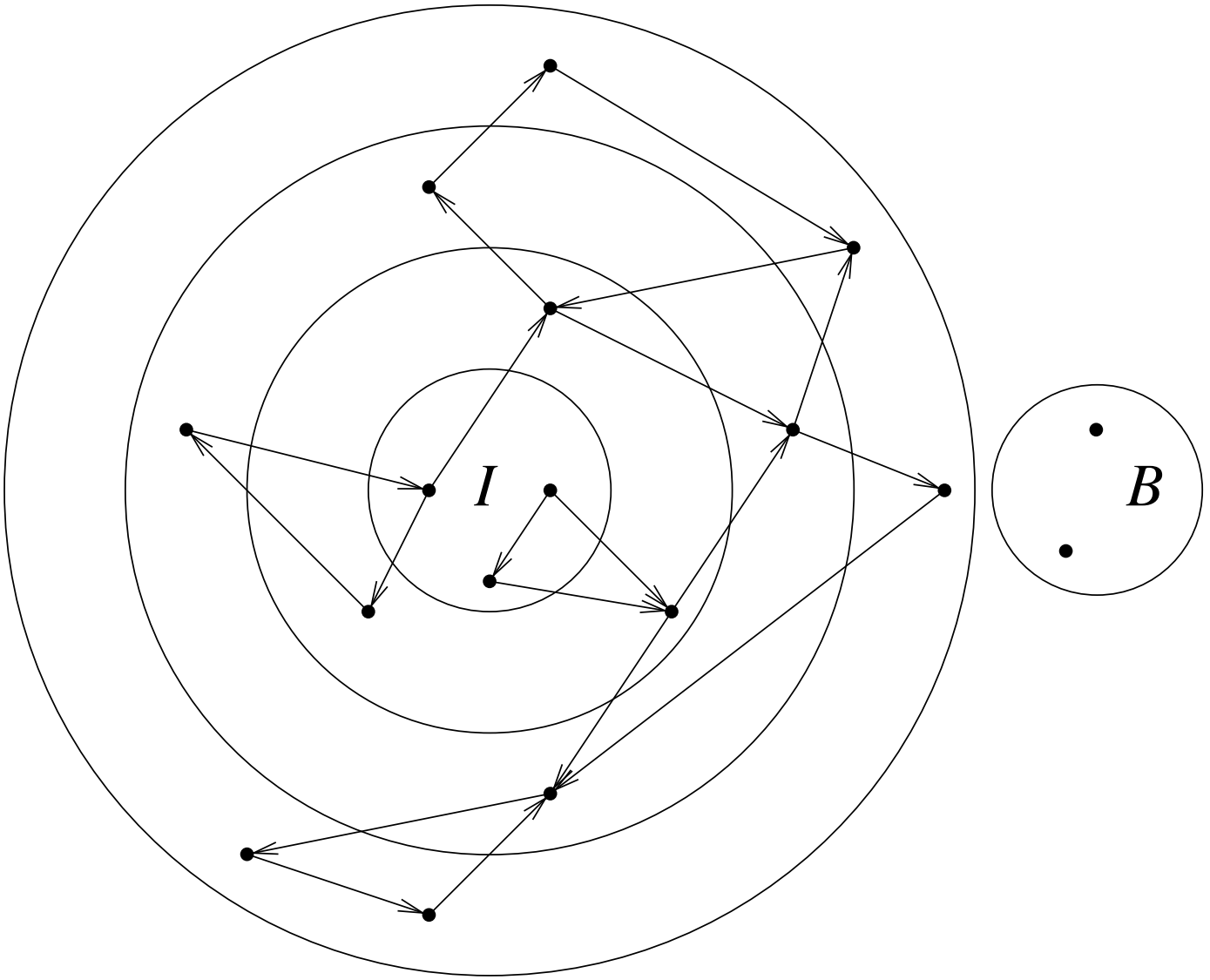
Forward Fixpoint Algorithm: Step 3



Forward Fixpoint Algorithm: Bad State Reached



Forward Fixpoint Algorithm: Termination, No Bad State Reachable



Forward Least Fixpoint Algorithm for Model Checking Safety

initial states I, transition relation T, bad states B

model-check$^{\mu}_{\text{forward}}$ (I, T, B)

$S_C = \emptyset$; $S_N = I$;

while $S_C \neq S_N$ **do**

$S_C = S_N$;

if $B \cap S_C \neq \emptyset$ **then**

return “found error trace to bad states”;

$S_N = S_C \cup Img(S_C)$;

done;

return “no bad state reachable”;

symbolic model checking represents set of states in this BFS symbolically

BDDs as Symbolic Representation

- BDDs are canonical representation for boolean functions
  - states encoded as bit vectors $\in \mathbb{B}^n$
  - set of states $S \subseteq \mathbb{B}^n$ as BDDs for characteristic function $f_S: \mathbb{B}^n \rightarrow \mathbb{B}$

$$f_S(s) = 1 \quad \Leftrightarrow \quad s \in S$$

- for all *set operations* there are linear BDD operations
  - except for Img which is exponential (often also in practice)

$$s \in Img(f) \quad \Leftrightarrow \quad \exists\, t \in \mathbb{B}^n [f(s) \wedge T(s,t)]$$

- variable ordering has strong influence on size of BDDs
- conjunctive partitioning of transition relation is a must

Termination Check in Symbolic Reachability is in QBF

- checking $S_C = S_N$ in 2nd iteration results in QBF decision problem

$$\forall s_0, s_1, s2[I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \rightarrow I(s_2) \vee \exists t_0[I(t_0) \wedge T(t_0, s_2)]]$$

- not **eliminating quantifiers** results in QBF with one alternation
  - note: number of necessary iterations bounded by 2^n

- circuit reachability is PSPACE complete [Savitch'70]

$$T^{2 \cdot i}(s,t) :\equiv \exists\, m[\forall c[\exists\, l, r[(c \rightarrow (l,r) = (s,m)) \wedge (\overline{c} \rightarrow (l,r) = (m,t)) \wedge T^i(l,r)]]]$$

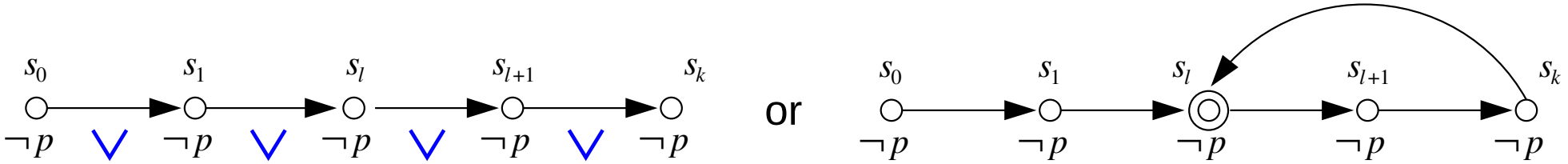
- so why not forget about termination and concentrate on bug finding?

  $\Rightarrow$ Bounded Model Checking

Bounded Model Checking (BMC)

[BiereCimattiClarkeZhu TACAS'99]

- look only for counter example made of k states (the bound)



- simple for safety properties $\mathbf{G}p$ (e.g. $p = \neg B$)

$$I(s_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})\right) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

- harder for liveness properties $\mathbf{F}p$

$$I(s_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})\right) \wedge \left(\bigvee_{l=0}^{k} T(s_k, s_l)\right) \wedge \bigwedge_{i=0}^{k} \neg p(s_i)$$

Bounded Model Checking State-of-the-Art

- increase in efficiency of SAT solvers (i.e. zChaff) helped a lot
- SAT more robust than BDDs in bug finding

  (shallow bugs are easily reached by explicit model checking or testing)
- better unbounded but still SAT based model checking algorithms
  - see for instance invited talk by Ken McMillan at SAT'04 in Vancouver
- 3rd Intl. Workshop on Bounded Model Checking (BMC'05)

  (in exactly 3 weeks, almost same place)
- other logics and better encodings

Original Translation for LTL and Lasso Witnesses

[BiereCimattiClarkeZhu TACAS'99]

on 1st look seems exponential
(in formula size $|f|$)

on 2nd look cubic
(in k and linear in $|f|$)

on 3rd look quadratic
(associativity)

on 4th look linear
(adhoc simplifications)

but binary operators $\mathbf{U}$, $\mathbf{R}$
make it at least quadratic again

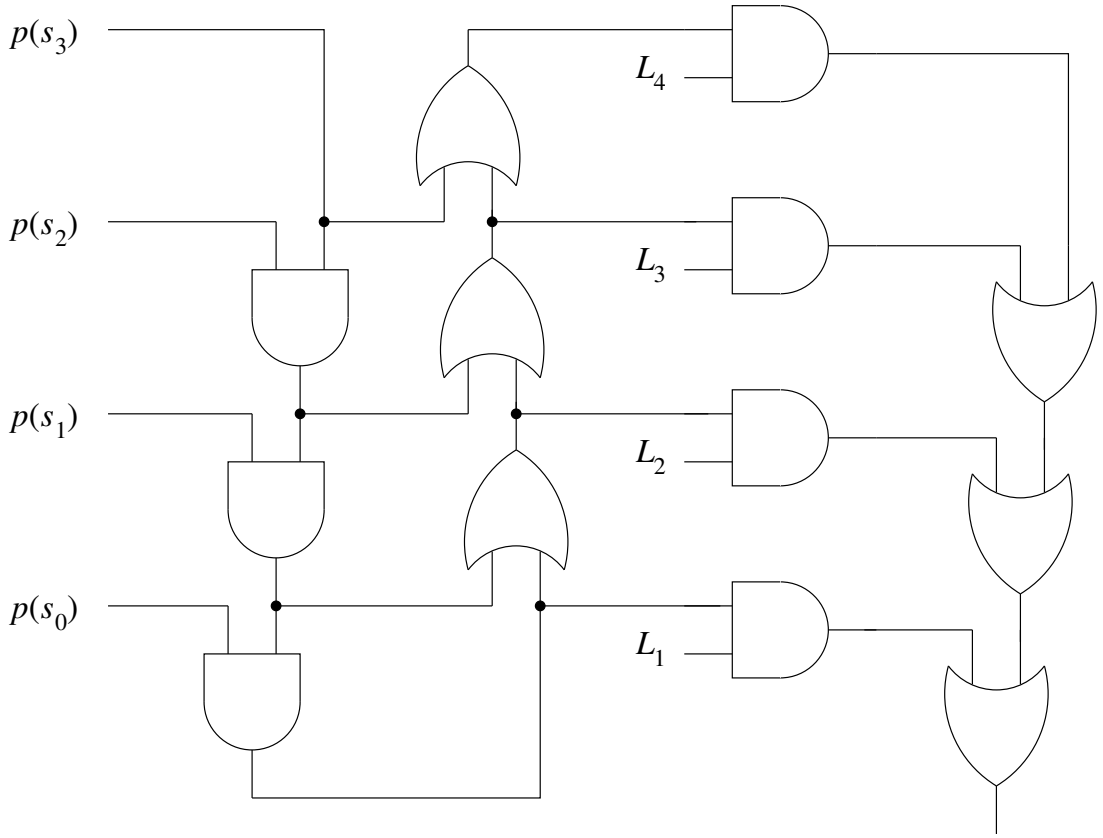
$$
\begin{aligned}
{}_l[p]^i_k &:= p(s_i) \\
{}_l[\neg p]^i_k &:= \neg p(s_i) \\
{}_l[f \wedge g]^i_k &:= {}_l[f]^i_k \wedge {}_l[g]^i_k \\
{}_l[\mathbf{X}\ f]^i_k &:= {}_l[f]^{next(i)}_k \\
{}_l[\mathbf{G}\ f]^i_k &:= \bigwedge_{j=min(l,i)}^{k} {}_l[f]^j_k \\
{}_l[\mathbf{F}\ f]^i_k &:= \bigvee_{j=min(l,i)}^{k} {}_l[f]^j_k
\end{aligned}
$$

with

$$
next(i) := \begin{cases} i+1 & \text{if } i < k \\ l & \text{else} \end{cases}
$$

Linear Circuit for Counterexample to Infinitely Often

original translation of $\mathbf{FG}p$ after applying associativity and sharing



(could be further simplified)

Simple and Linear Translation for LTL

[LatvalaBiereHeljankoJunttila FMCAD'04]

evaluate semantics on loop in two iterations

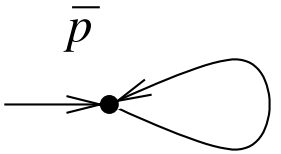
$\langle\ \rangle$ = 1st iteration $\quad [\ ]$ = 2nd iteration

$:=$	$i < k$	$i = k$
$[p]_i$	$p(s_i)$	$p(s_k)$
$[\neg p]_i$	$\neg p(s_i)$	$\neg p(s_k)$
$[\mathbf{X} f]_i$	$[f]_{i+1}$	$\bigvee_{l=0}^{k} (T(s_k, s_l) \wedge [f]_l)$
$[\mathbf{G} f]_i$	$[f]_i \wedge [\mathbf{G} f]_{i+1}$	$\bigvee_{l=0}^{k} (T(s_k, s_l) \wedge \langle \mathbf{G} f \rangle_l)$
$[\mathbf{F} f]_i$	$[f]_i \vee [\mathbf{F} f]_{i+1}$	$\bigvee_{l=0}^{k} (T(s_k, s_l) \wedge \langle \mathbf{F} f \rangle_l)$
$\langle \mathbf{G} f \rangle_i$	$[f]_i \wedge \langle \mathbf{G} f \rangle_{i+1}$	$[f]_k$
$\langle \mathbf{F} f \rangle_i$	$[f]_i \vee \langle \mathbf{F} f \rangle_{i+1}$	$[f]_k$

Simple and Linear Translation for LTL cont.

- semantic of LTL on *single path* is the same as CTL semantic
  - symbolically implement fixpoint calculation for (A)CTL
  - fixpoint computation terminates after 2 iterations (not k)
  - boolean fixpoint equations $\Rightarrow$ boolean graphs
- easy to implement and optimize, fast
  - generalized to past time [LatvalaBiereHeljankoJunttila VMCAI'05]
  - minimal counter examples for past time [SchuppanBiere TACAS'05]
  - incremental (and complete) [LatvalaHeljankoJunttila CAV'05]

Why Not Just Try to Satisfy Boolean Equations directly?



recursive expansion $\mathbf{F}p \equiv p \vee \mathbf{XF}p$

checking $\mathbf{G}\overline{p}$ implemented as search for witness for $\mathbf{F}p$

Kripke structure: single state with self loop in which p does not hold

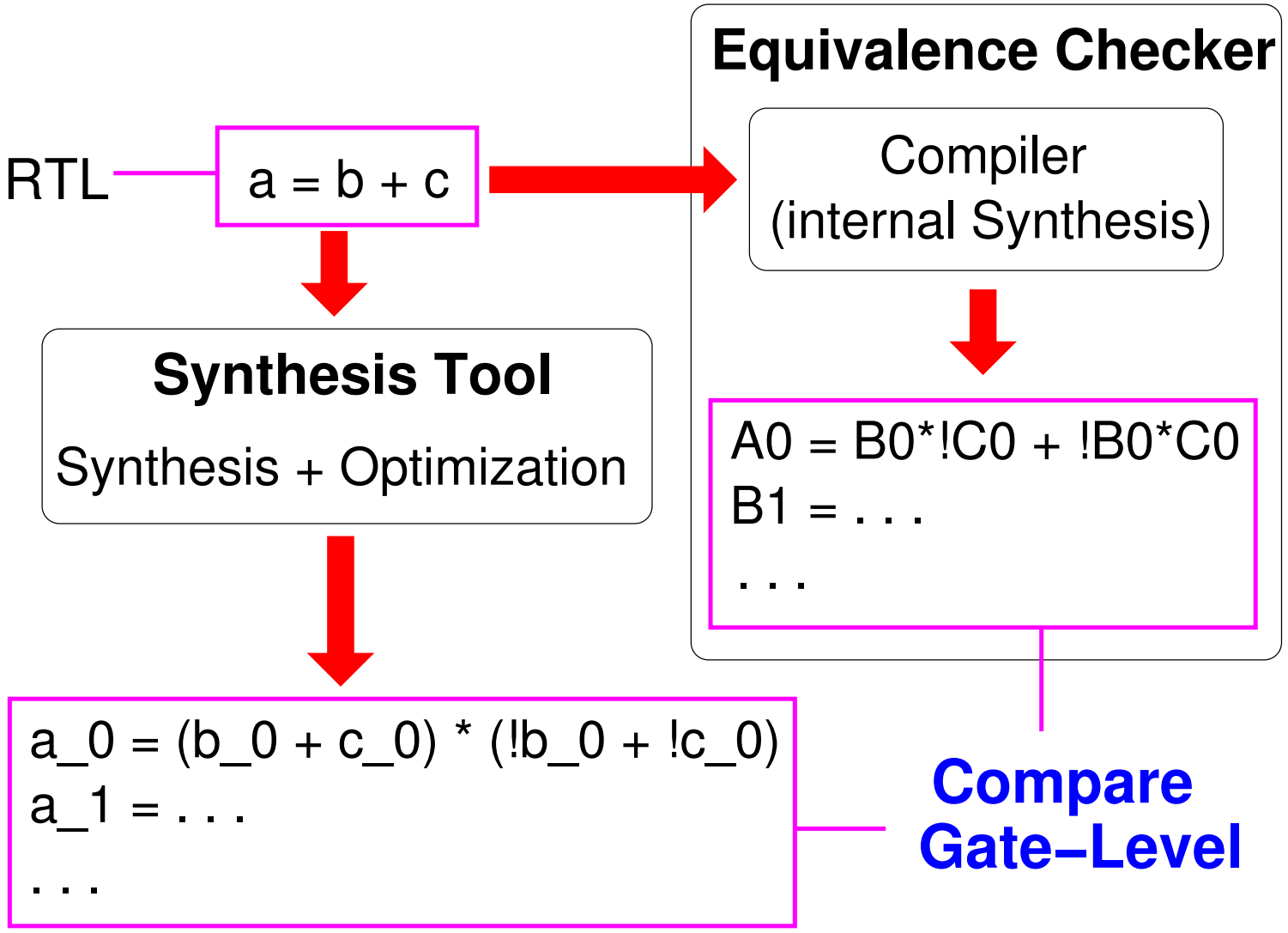
incorrect translation of $\mathbf{F}p$:

$$\overbrace{I(s_0) \wedge T(s_0, s_0)}^{\text{model constraints}} \wedge \underbrace{([\mathbf{F}p] \leftrightarrow p(s_0) \vee [\mathbf{F}p])}_{\text{translation}} \wedge \overbrace{[\mathbf{F}p]}^{\text{assumption}}_{x}$$

since it is satisfiable by setting $x = 1$ though $p(s_0) = 0$

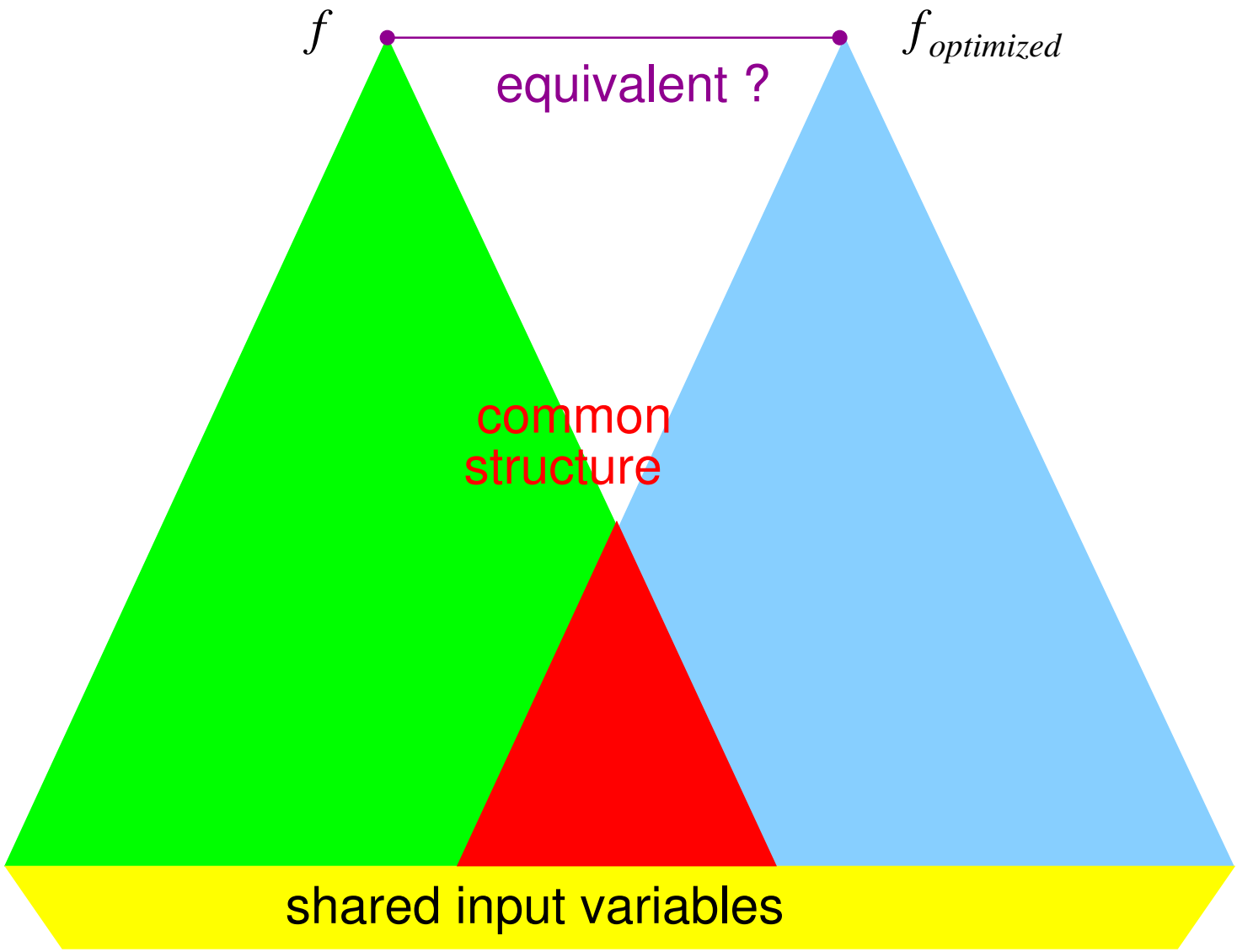
(x fresh boolean variable introduced for $[\mathbf{F}p]$)

Equivalence Checking

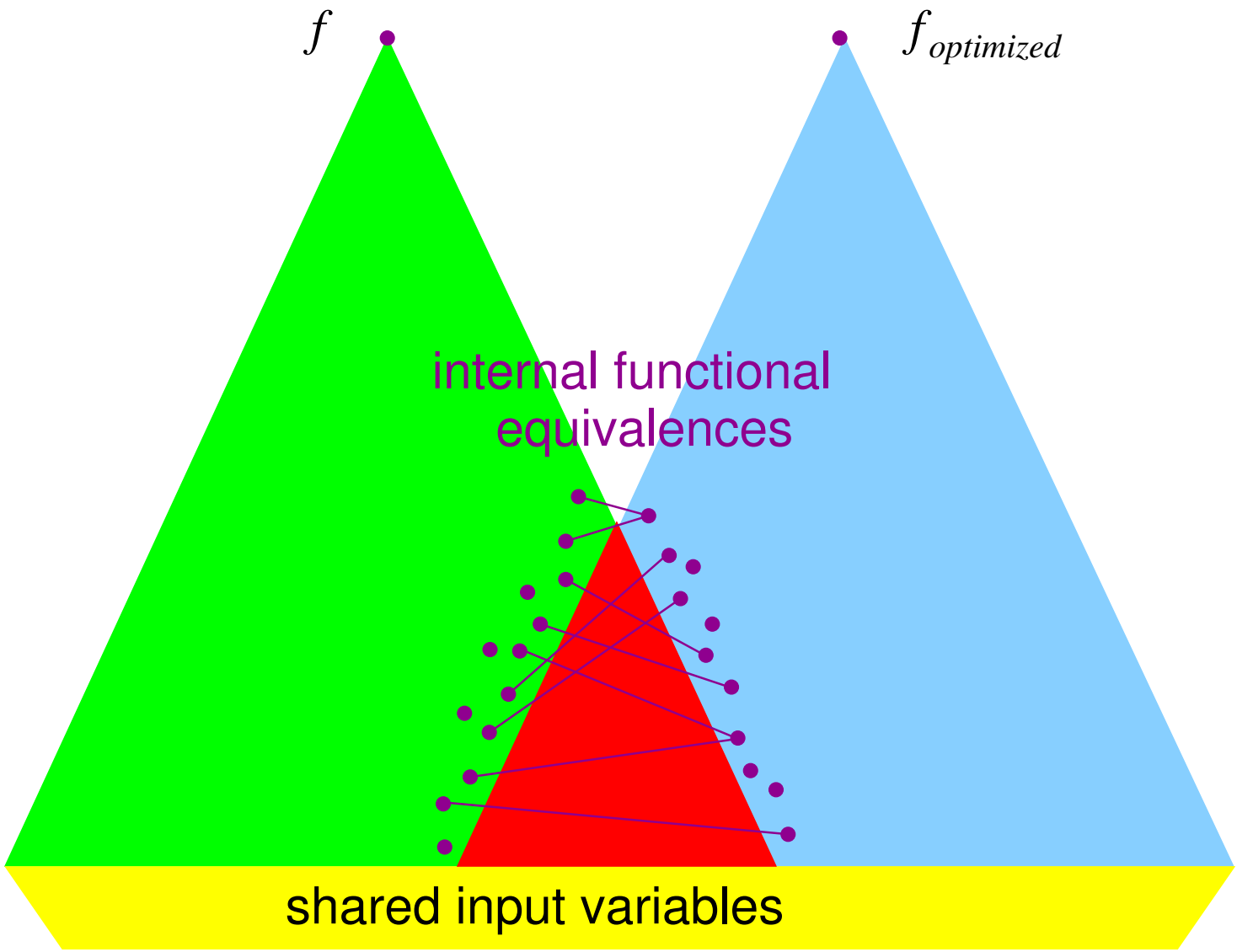


(RTL = Register Transfer Level)

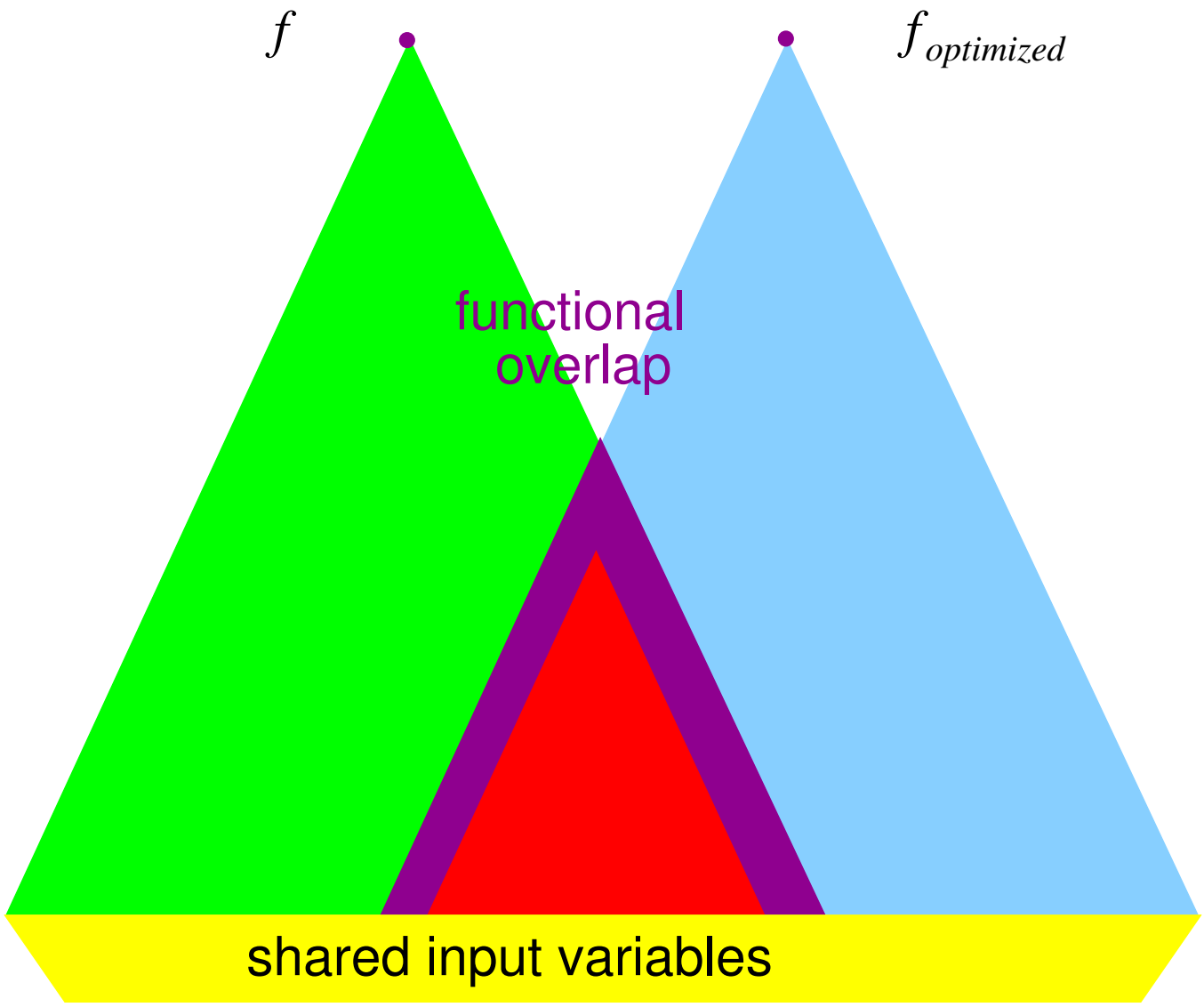
Equivalence Checking in the Large



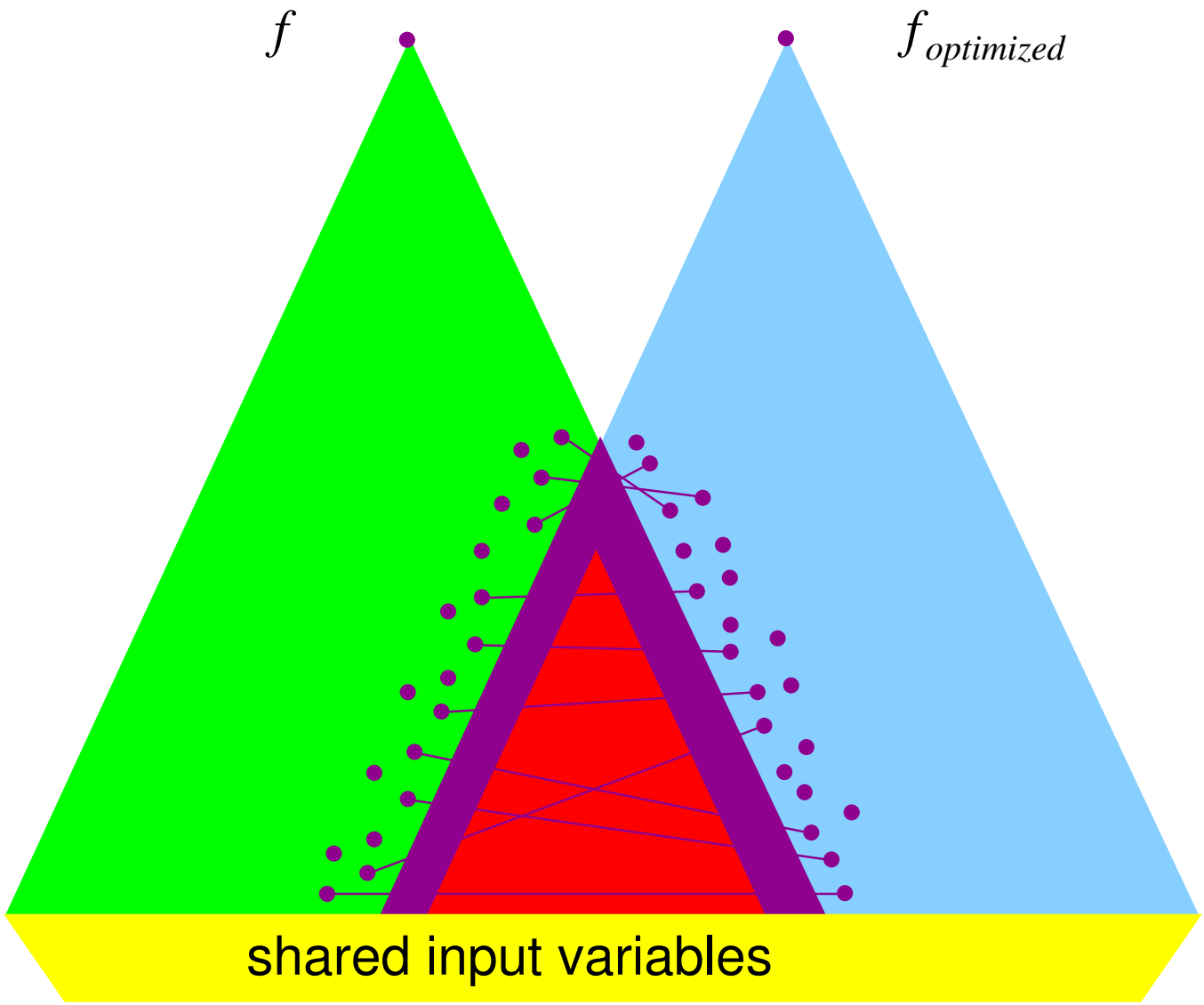
Equivalence Checking in the Large



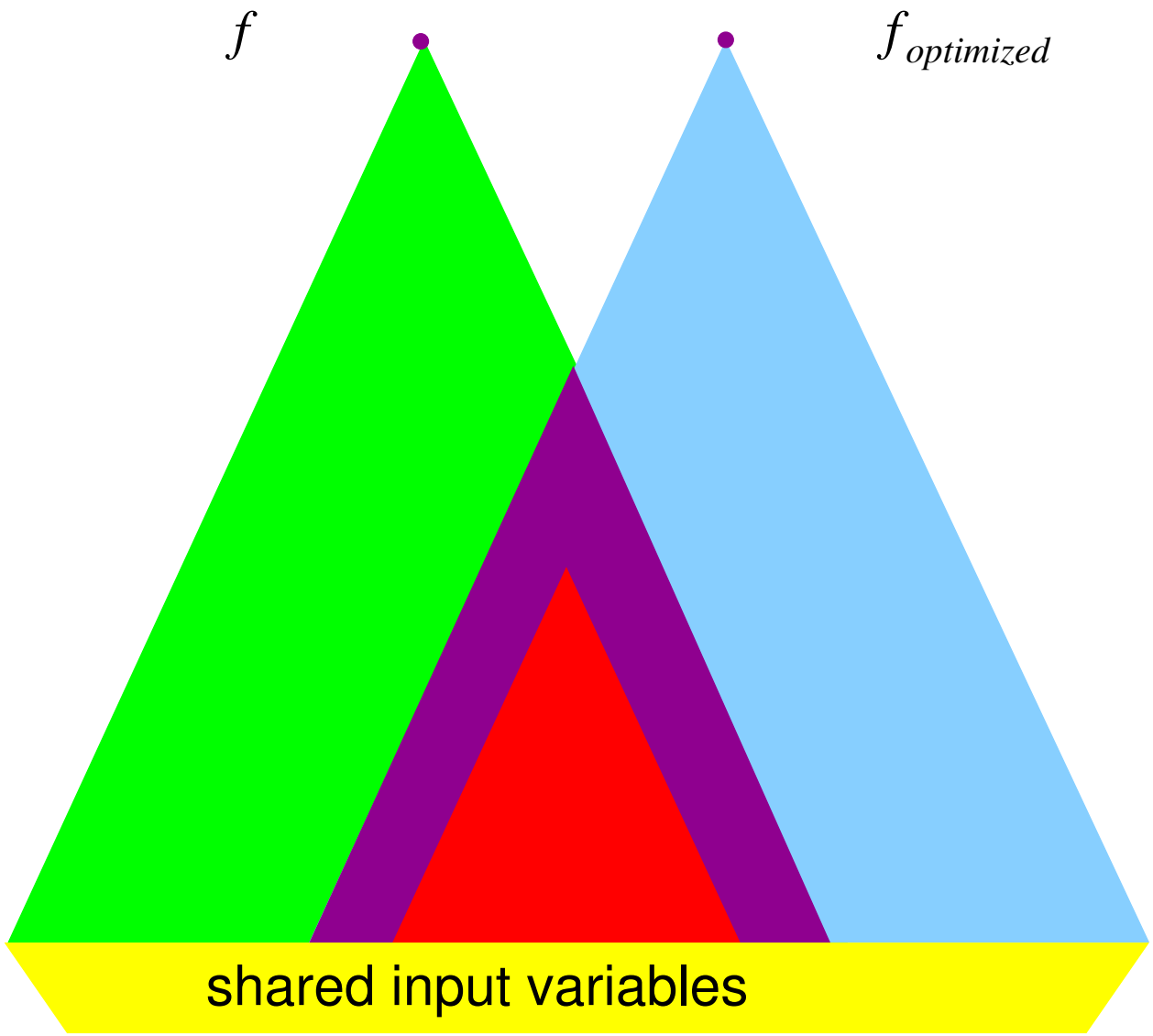
Equivalence Checking in the Large



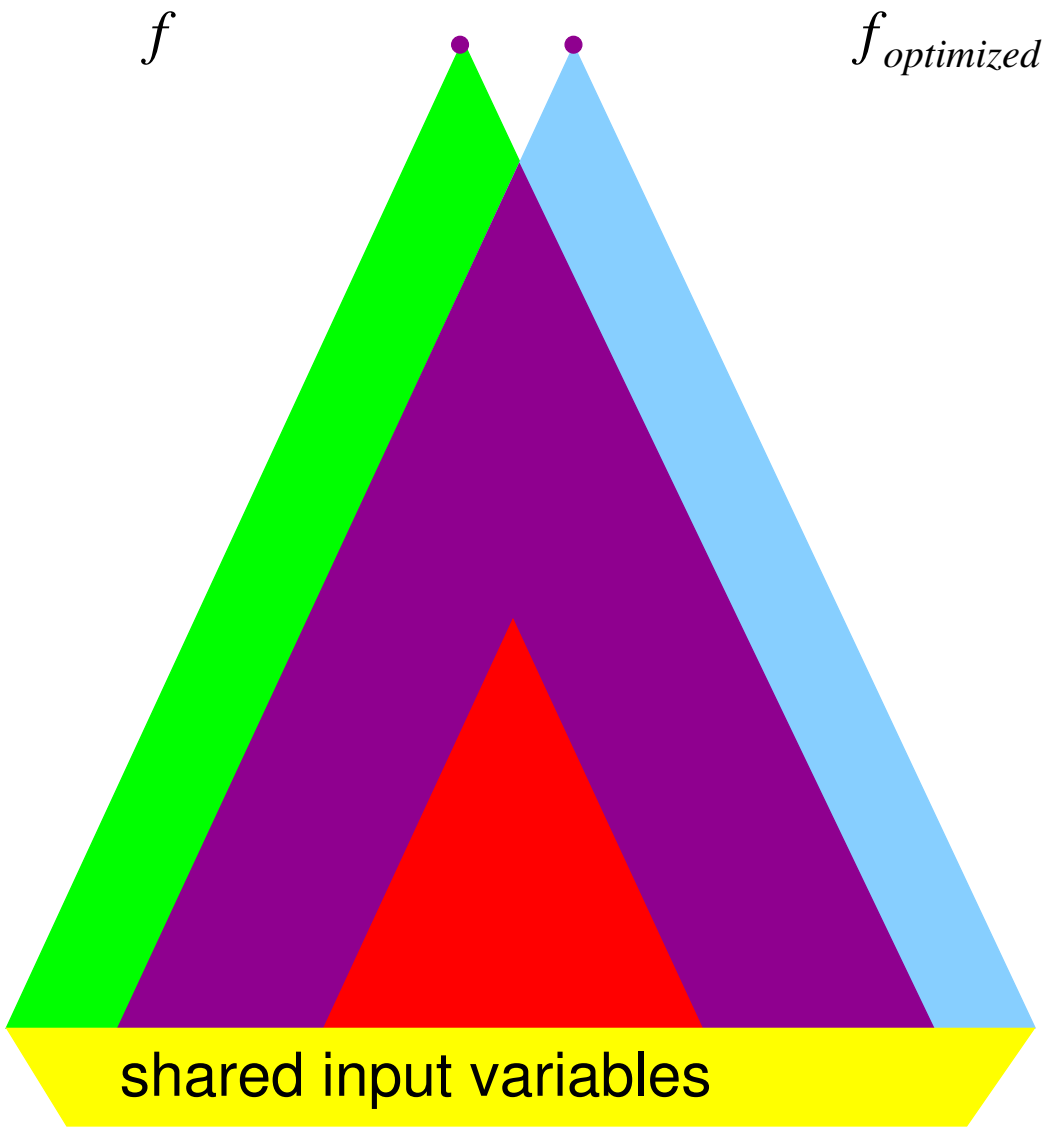
Equivalence Checking in the Large



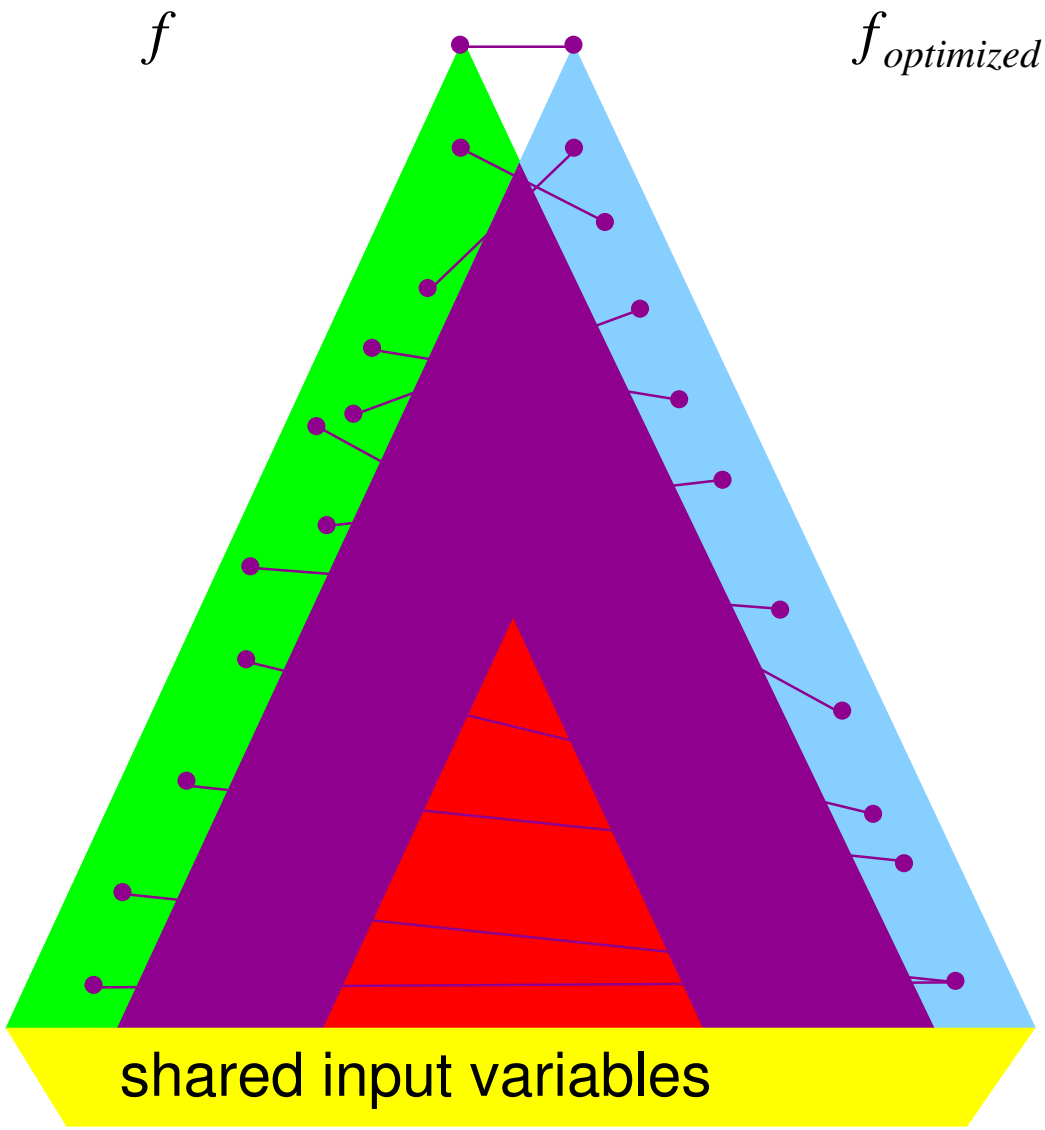
Equivalence Checking in the Large



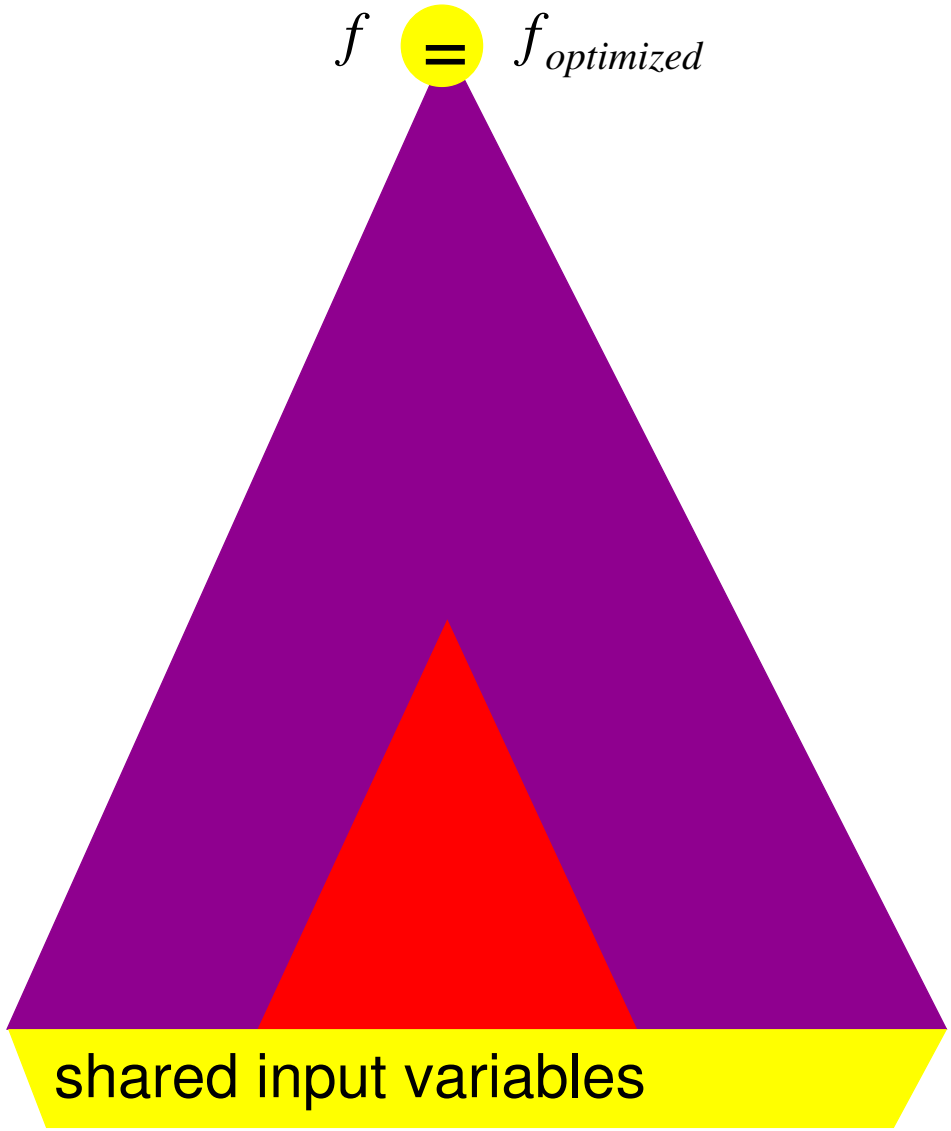
Equivalence Checking in the Large



Equivalence Checking in the Large



Equivalence Checking in the Large



Equivalence Checking

- BDD-Sweeping [KühlmannKrohm DAC'97]
  - levelized, resource driven construction of small overlapping BDDs
  - BDDs are mapped back to circuit nodes
  - circuit nodes with same BDD are functionally equivalent
- can be combined with top-down approach (e.g. backward chaining)
  - interleave BDD building with circuit based SAT solver
- recently SAT-Sweeping [Kühlmann ICCAD'04]
  - candidate pairs of equivalent circuit nodes through random simulation
  - more robust than BDDs, particularly when used as simplifier for BMC

Automatic Test Pattern Generation (ATPG)

- need to test chips **after** manufacturing
  - manufacturing process introduces faults ($<$ 100% yield)
  - faulty chips can not be sold (should not)
  - generate all test patterns from functional logic description

- simplified failure model
  - at most one wire has a fault
  - fault results in fixing wire to a logic constant:

    “stuck at zero fault” (s-a-0) “stuck at one fault” (s-a-1)

ATPG with D-Algorithm

[Roth'66]

- adding logic constants D and $\overline{D}$ allows to work with only one circuit

0	represents	0	in fault free and	0	in faulty circuit
1	represents	1	in fault free and	1	in faulty circuit
D	represents	1	in fault free and	0	in faulty circuit
$\overline{D}$	represents	0	in fault free and	1	in faulty circuit

- otherwise obvious algebraic rules (propagation rules)

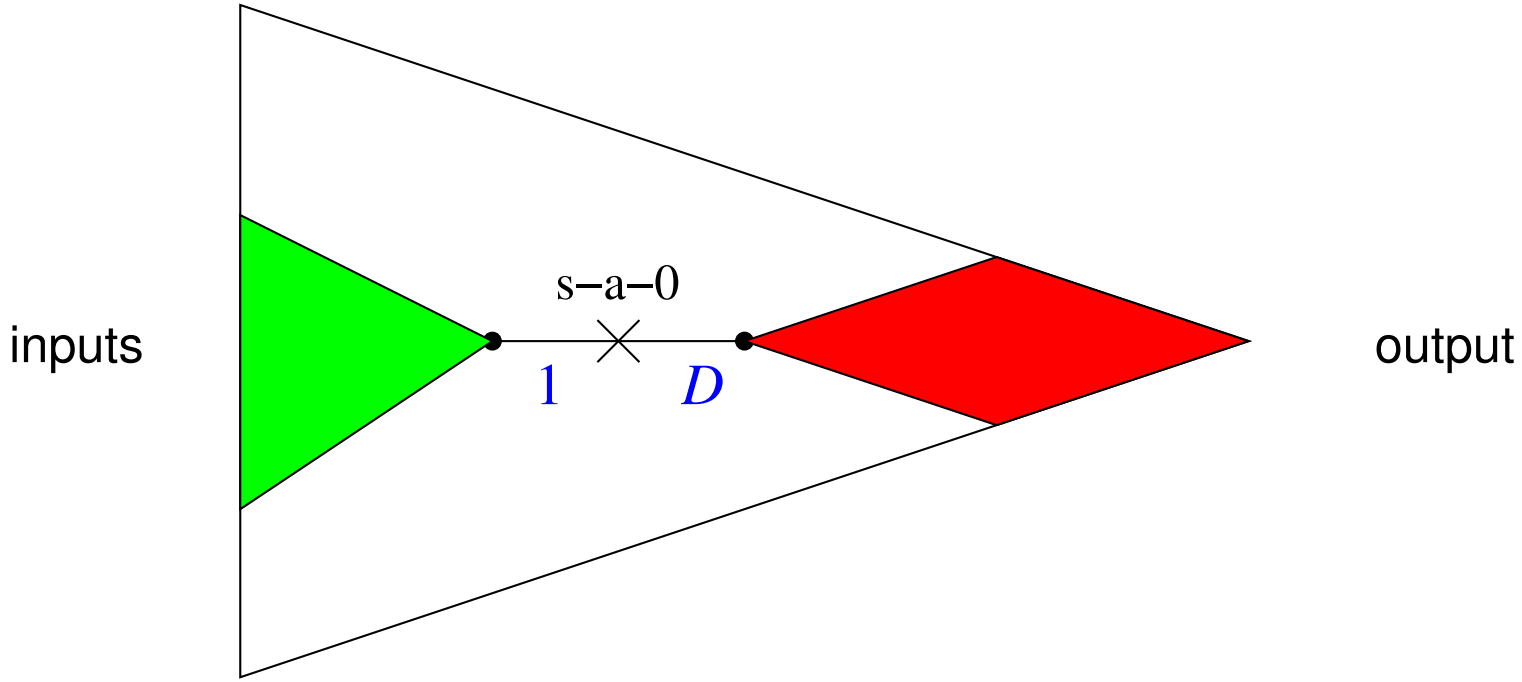
$$1 \wedge D \equiv D \qquad 0 \wedge D \equiv 0 \qquad \overline{D} \wedge D \equiv 0 \qquad \text{etc.}$$

- new conflicts: e.g. variable/wire can not be 0 and D at the same time

Fault Injection for S-A-0 Fault

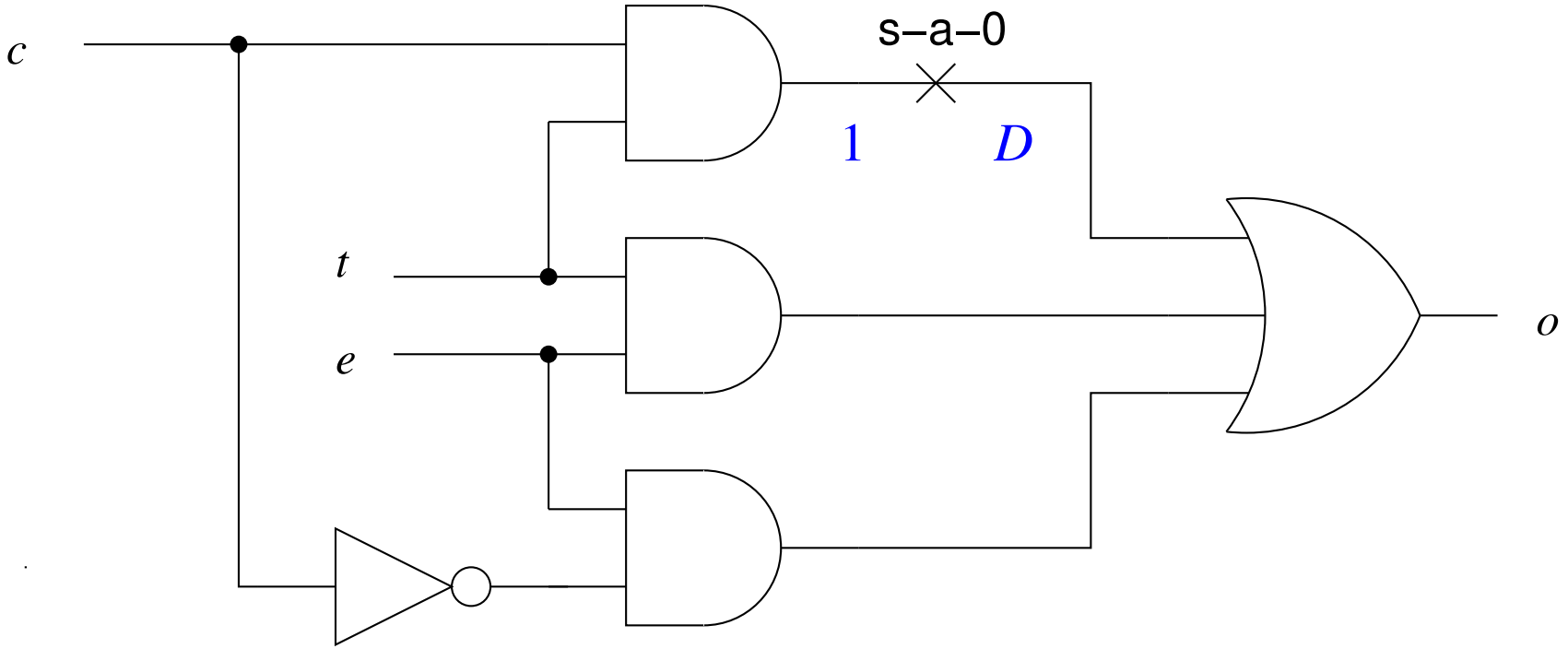
assume opposite value 1 before fault

(both for fault free and faulty circuit)

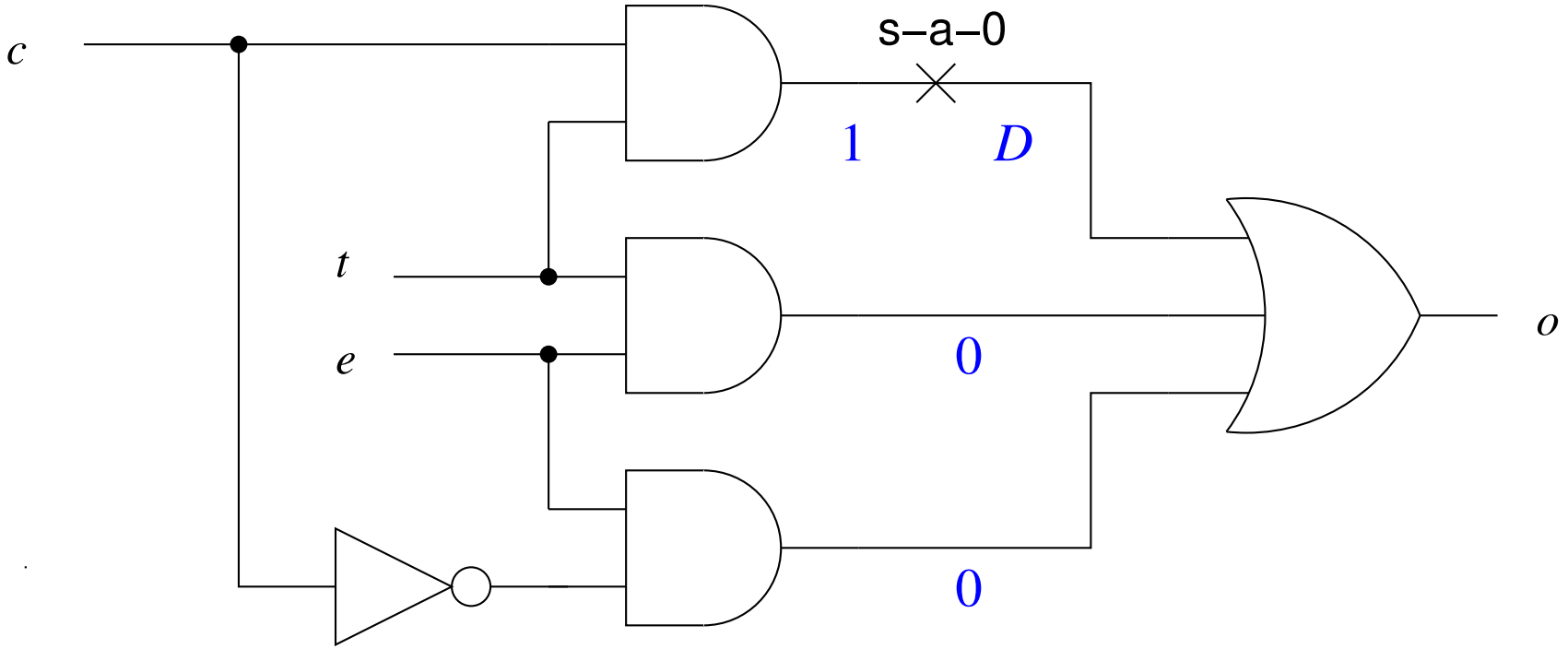


assume difference value D after fault

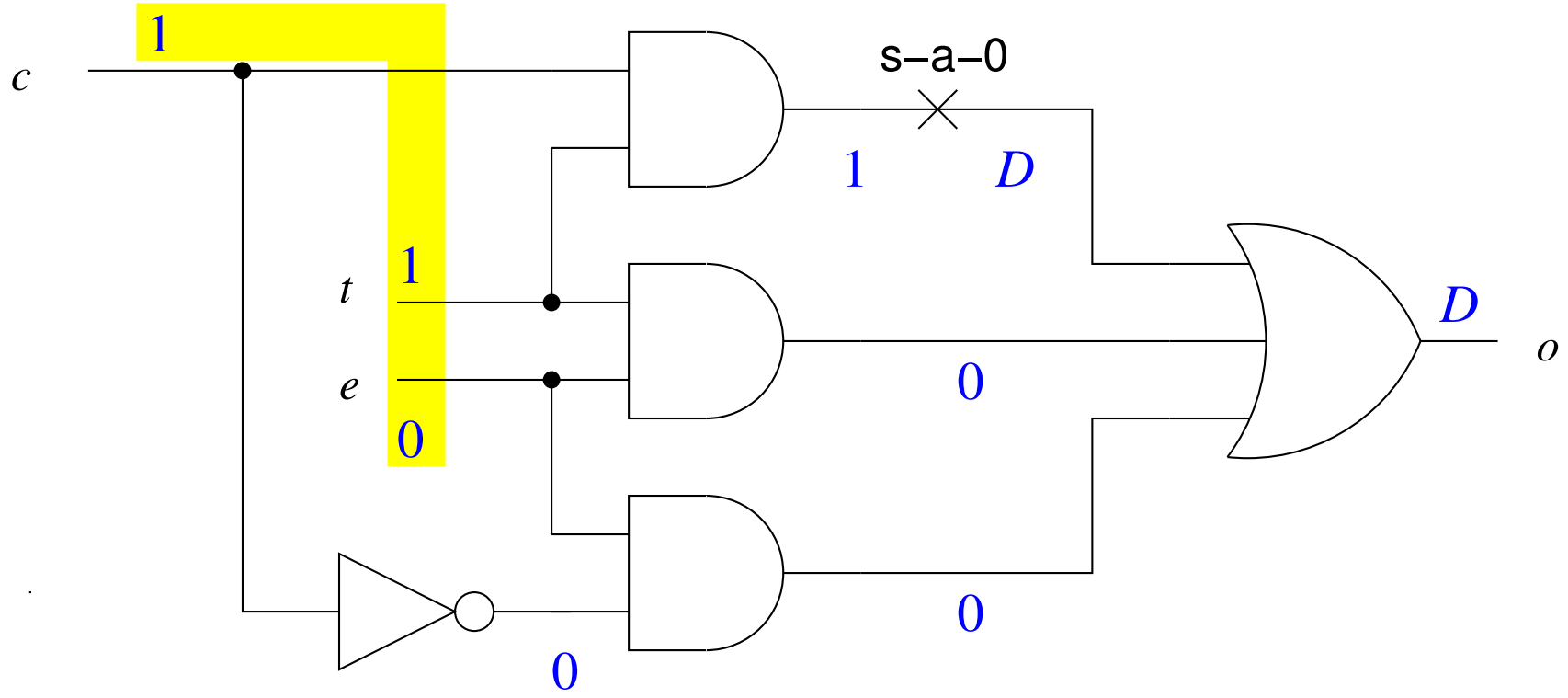
D-Algorithm Example: Fault Injection



D-Algorithm Example: Path Sensitation



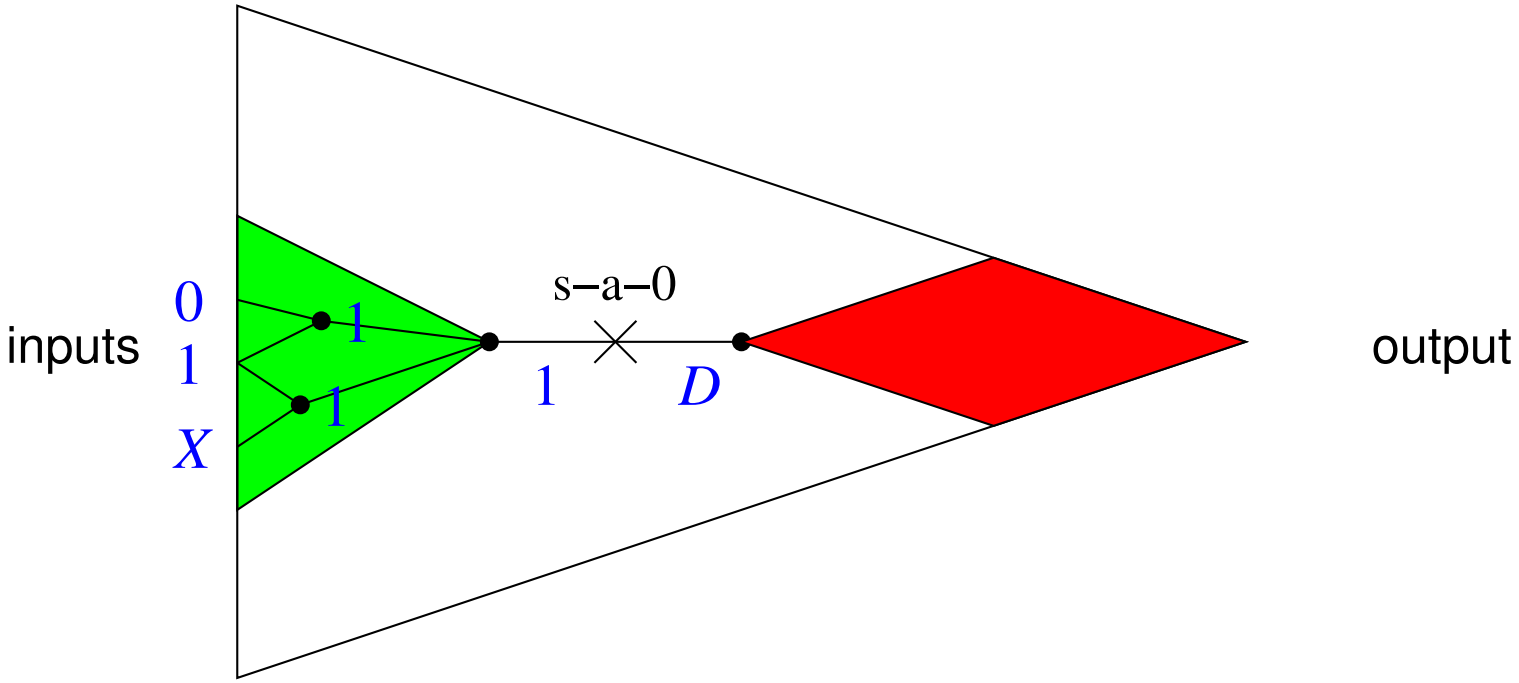
D-Algorithm Example: Propagation



test vector $(c,t,e) = (1,1,0)$

Justification

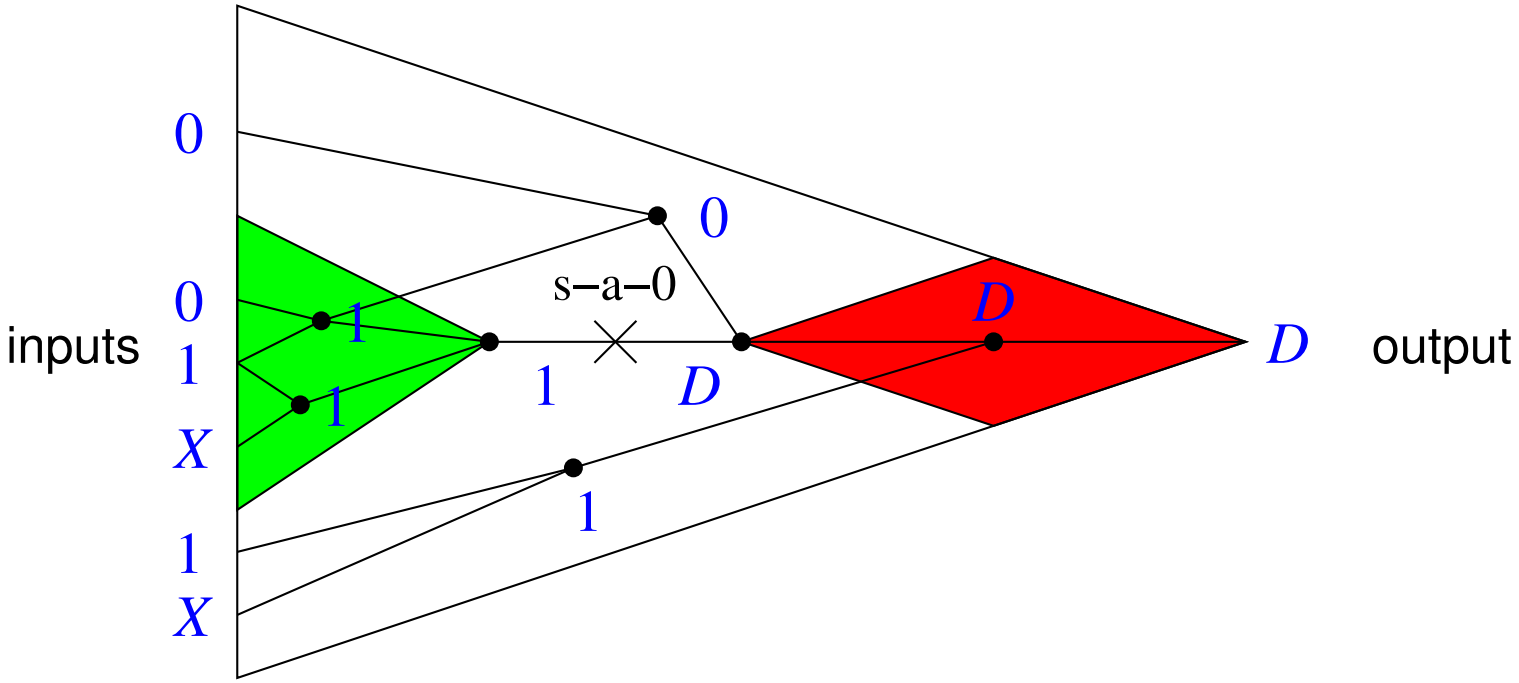
generate partial input vector to justify 1



only backward propagation, remaining unassigned inputs can be arbitrary

Observation

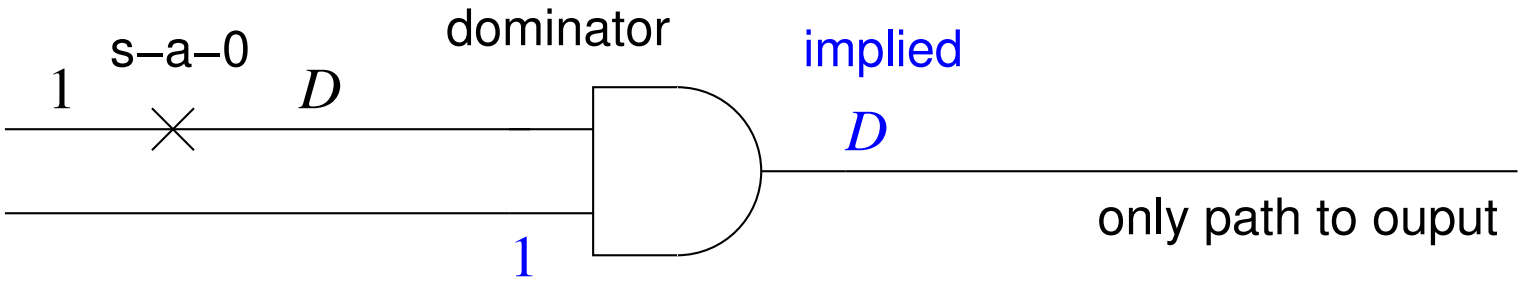
extend partial input vector to propagate D or $\overline{D}$ to ouput



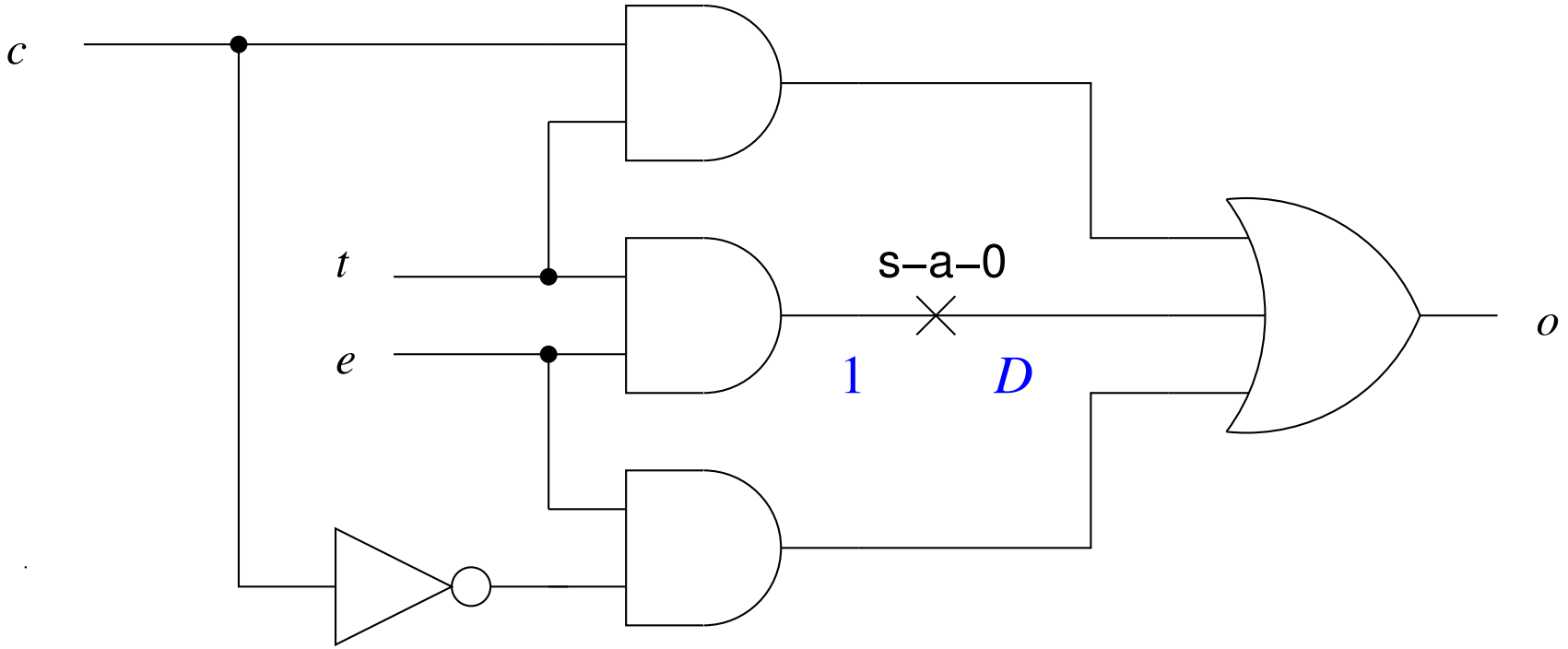
forward propagation of D and $\overline{D}$, backward propagation of 0 and 1

Dominators and Path Sensitation

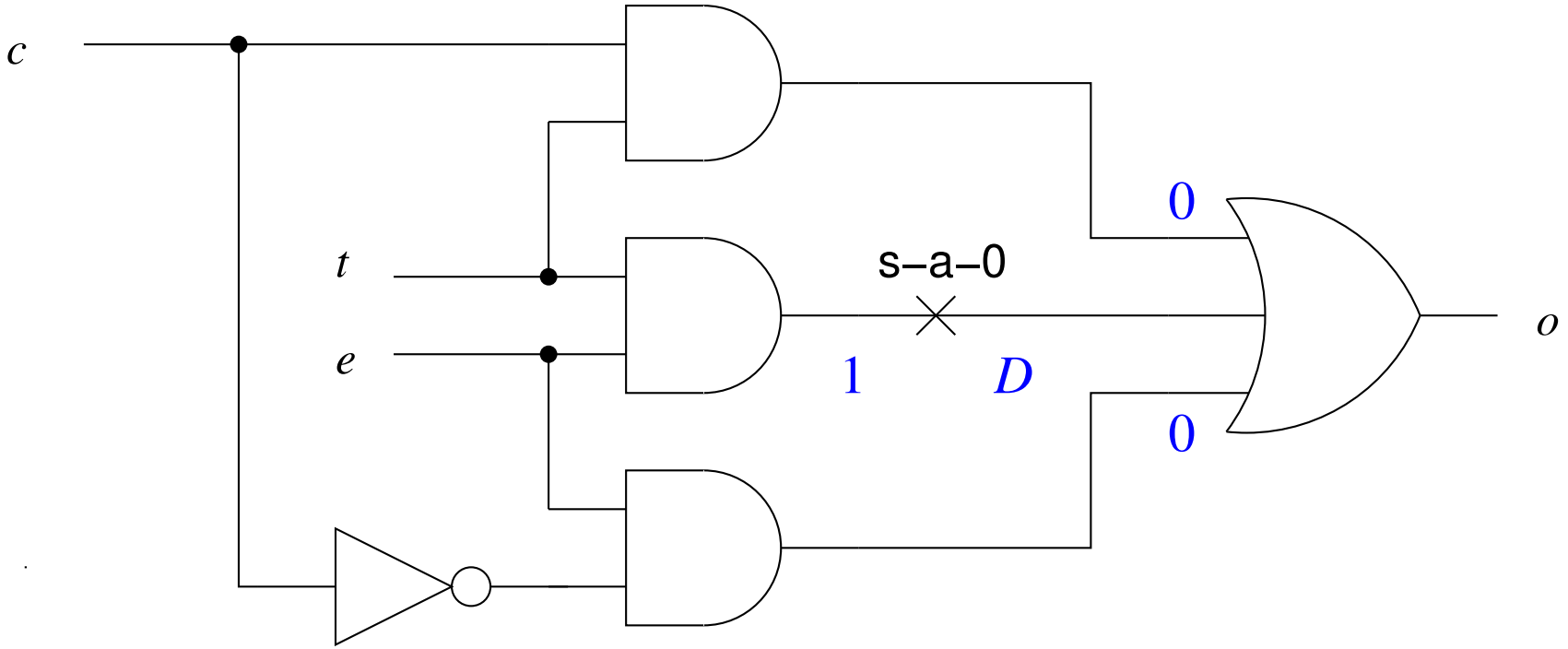
- idea: use circuit topology for additional necessary conditions
  - assign and propagate these conditions after fault injection
- gate dominates fault iff every path from fault to output goes through it
  - more exactly we determine wires (input to gates) that dominate a fault
- if input dominates a fault assign other inputs to non-controlling value



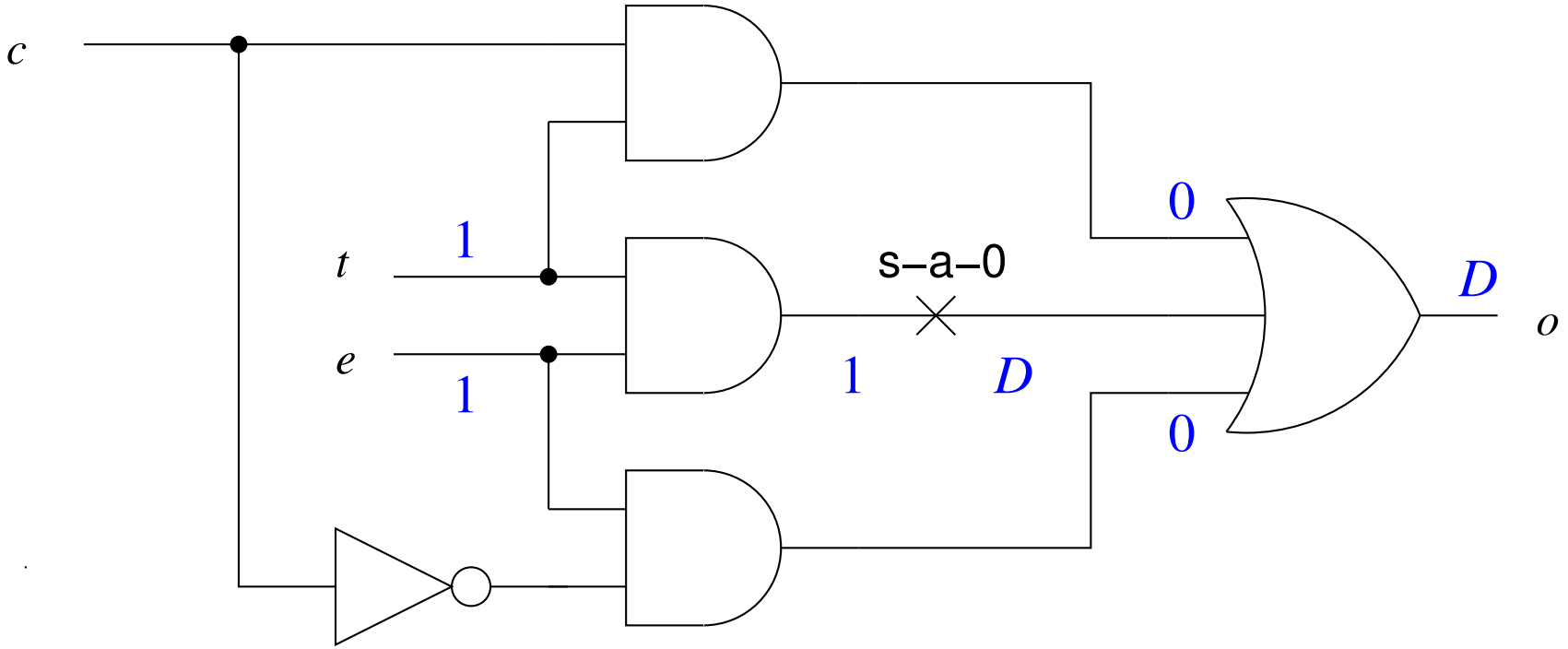
Redundancy Removal with D-Algorithm: Fault Injection



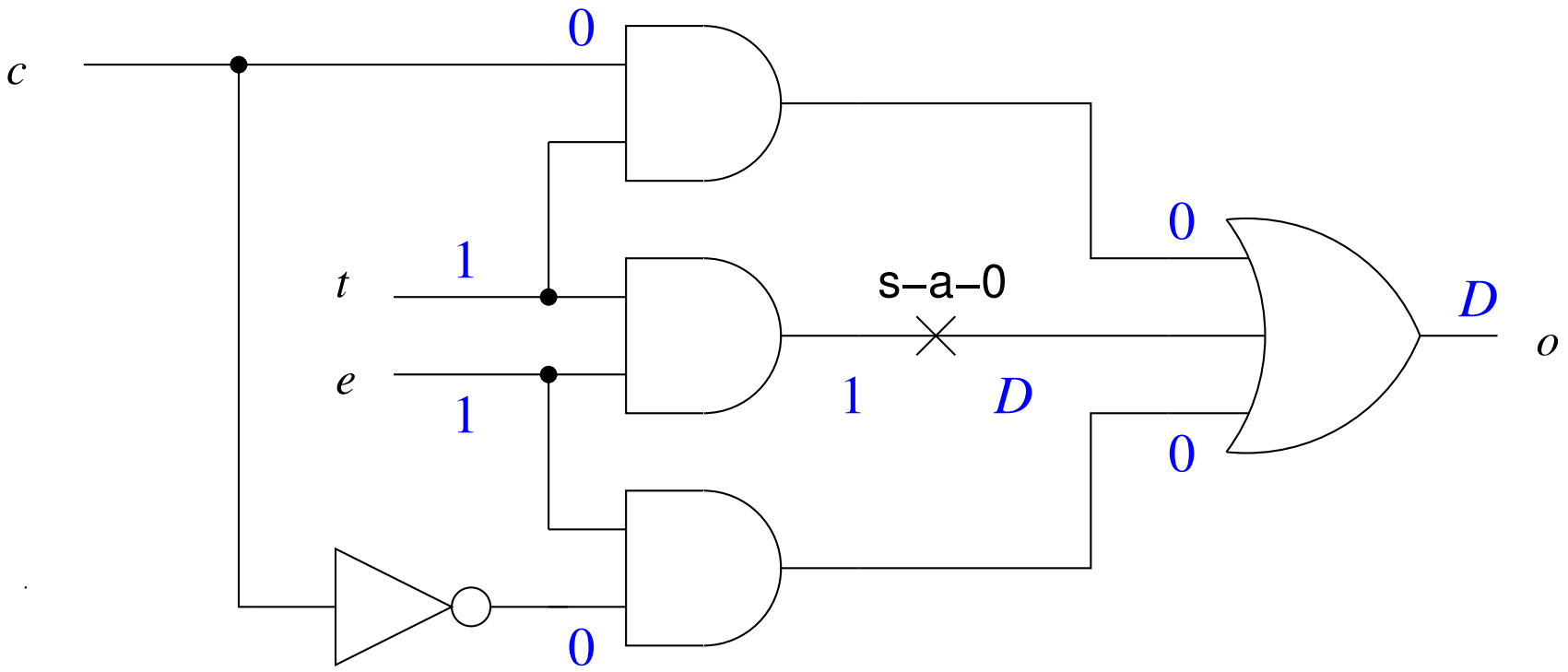
Redundancy Removal with D-Algorithm: Path Sensitation



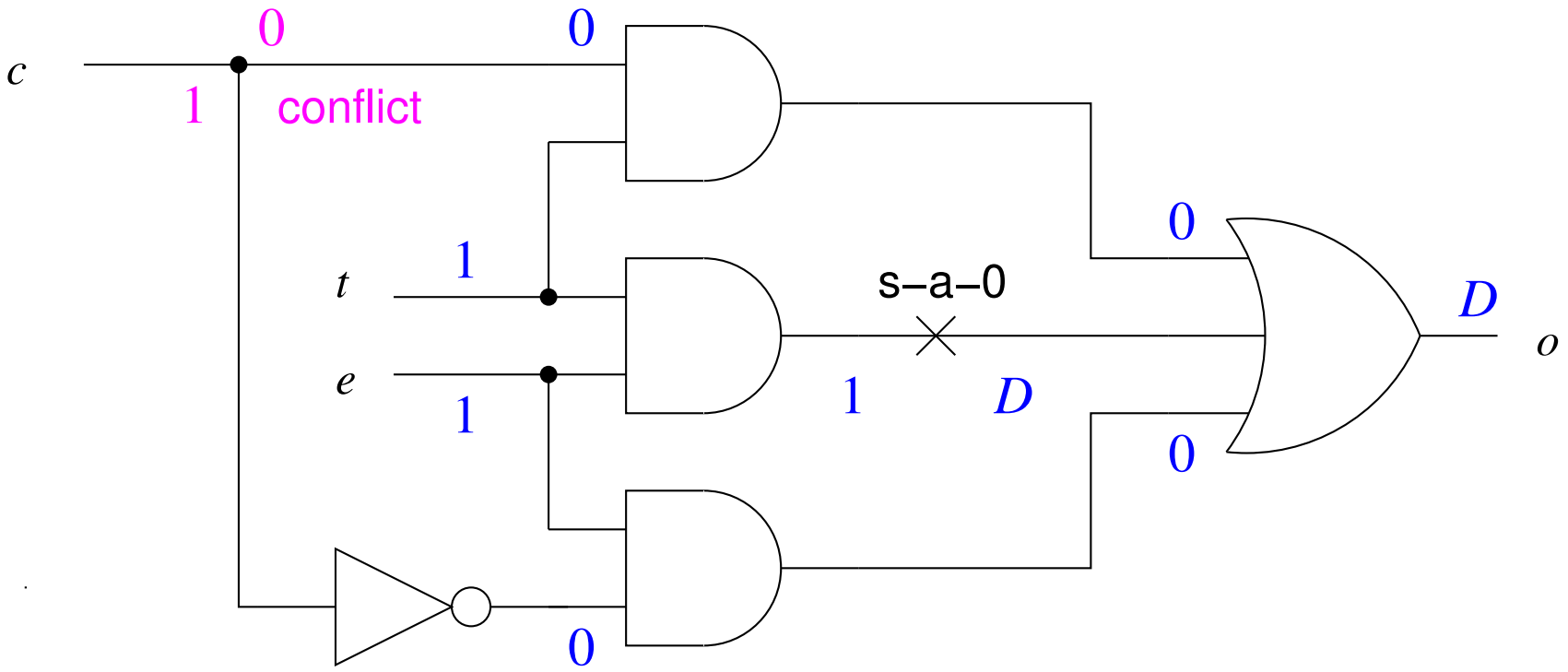
Redundancy Removal with D-Algorithm: 1st Propagation



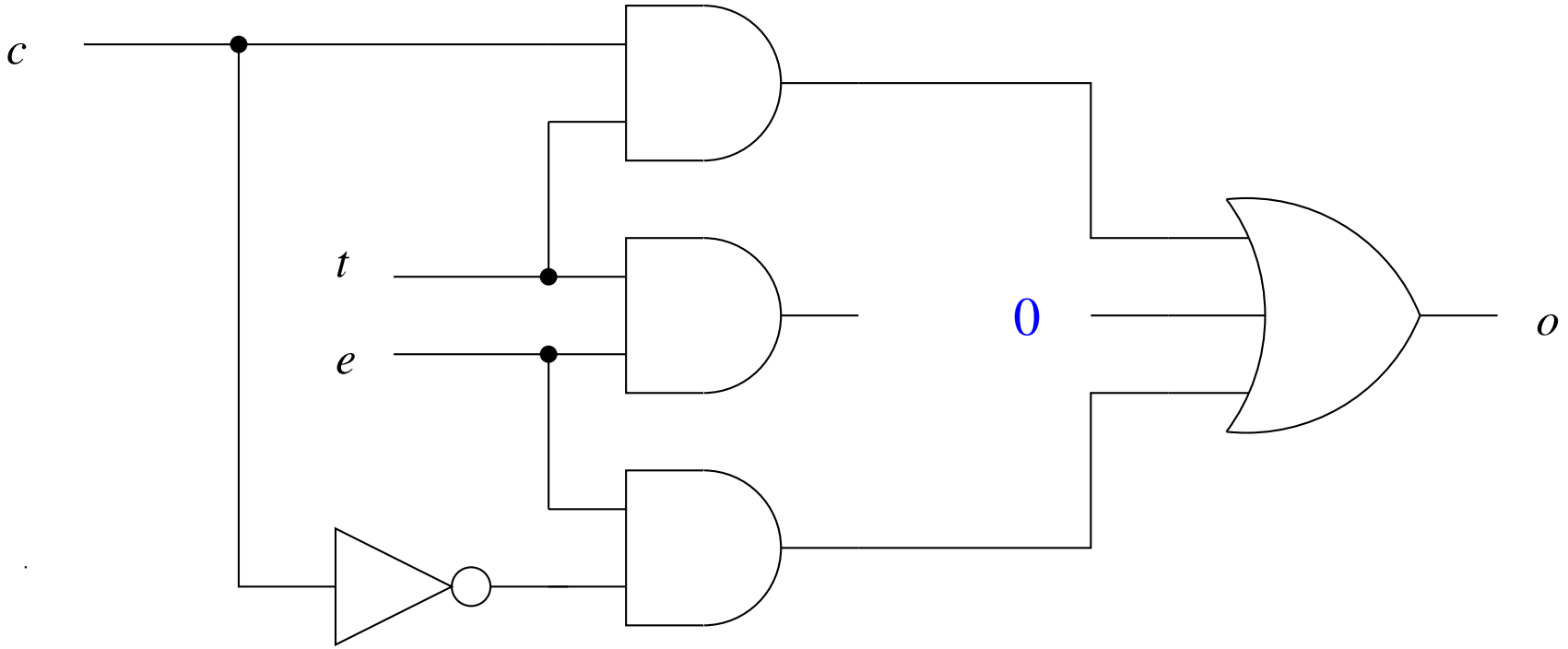
Redundancy Removal with D-Algorithm: 2nd Propagation



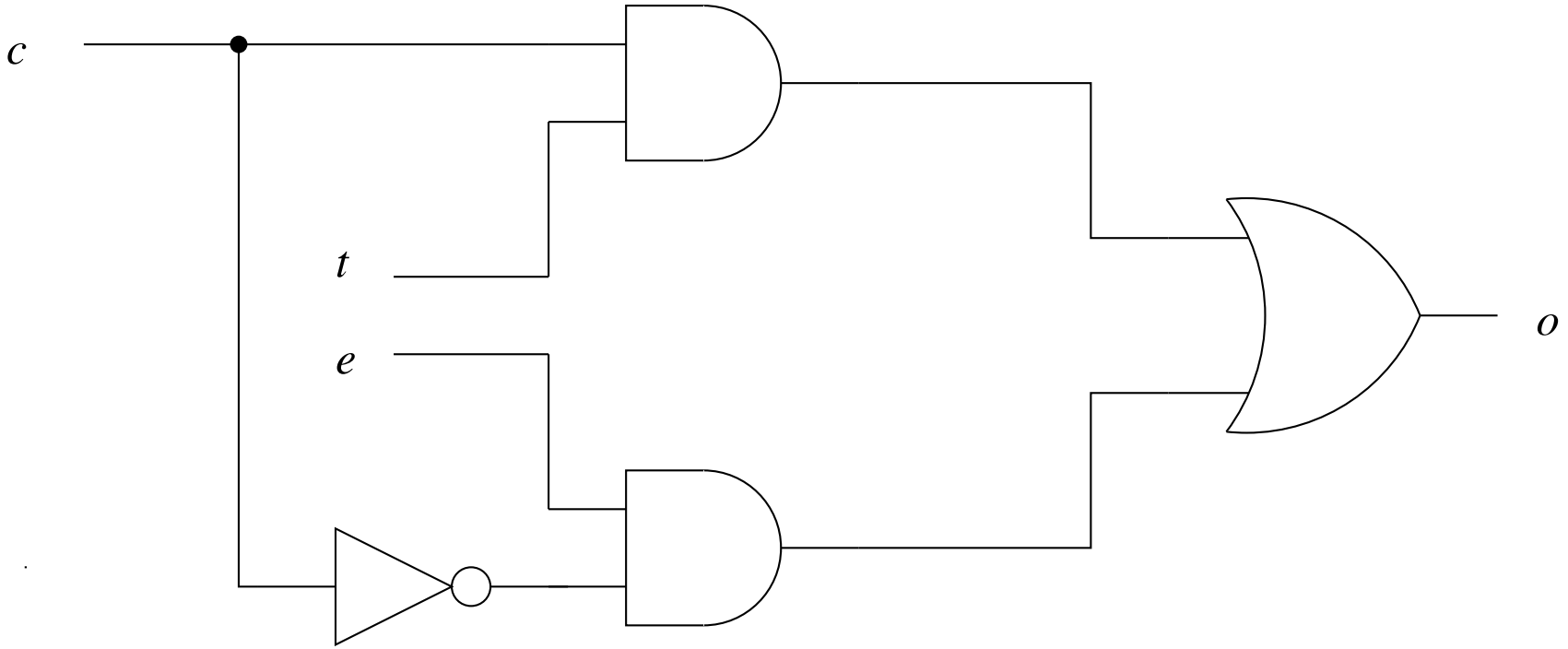
Redundancy Removal with D-Algorithm: Untestable



Redundancy Removal with D-Algorithm: Assume Fault



Redundancy Removal with D-Algorithm: Simplified Circuit



Redundancy Removal for SAT

- assume CNF is generated via Tseitin transformation from formula/circuit
  - formula = model constraints $+$ negation of property
  - CNF consists of gate input/output consistency constraints
  - plus additional unit forcing output o of whole formula to be 1
- remove redundancy in formula under assumption $o = 1$
- propagation of D or $\overline{D}$ to o does not make much sense
  - not interested in $o = 0$
  - check simply for unsatisfiability $\Rightarrow$ no need for D, $\overline{D}$ (!?)

Variable Instantiation

[AnderssonBjesseCookHanna DAC'02] and Oepir SAT solver

- satisfiability preserving transformation
- motivated by original pure literal rule:
  - if a literal l does not occur negatively in CNF f
  - then replace l by 1 in f (continue with $f[l \mapsto 1]$)
- generalization to variable instantiation:
  - if $f[l \mapsto 0] \rightarrow f[l \mapsto 1]$ is valid
  - then replace l by 1 in f (continue with $f[l \mapsto 1]$)

Why is Variable Instantiation a Generalization of the Pure Literal Rule?

Let $f \equiv f' \wedge f_0 \wedge f_1$ with

f' l does not occur

f_0 l occurs negatively

f_1 l occurs positively

further assume (assumption of pure literal rule)

$f_0 \equiv 1$

then

$$f[l \mapsto 0] \quad \Leftrightarrow \quad f' \wedge f_1[l \mapsto 0] \quad \overset{!}{\Rightarrow} \quad f' \quad \Leftrightarrow \quad f[l \mapsto 1]$$

Variable Instantiation Implementation

We have

$$f[l \mapsto 1] \quad \Leftrightarrow \quad f' \wedge \underbrace{f_1[l \mapsto 1]}_{1} \wedge f_0[l \mapsto 1] \quad \Leftrightarrow \quad f' \wedge f_0[l \mapsto 1] \quad \Leftrightarrow \quad f' \wedge \overbrace{f_0[l \mapsto 1]}^{n} \bigwedge_{i=1} C_i$$

and since $f[l \mapsto 0] \Rightarrow f'$ we only need show the validity of

$$f[l \mapsto 0] \rightarrow \bigwedge_{i=1}^{n} C_i$$

which is equivalent to the unsatisfiability of

$$f[l \mapsto 0] \wedge \overline{C_i} \qquad \text{for } i = 1 \ldots n$$

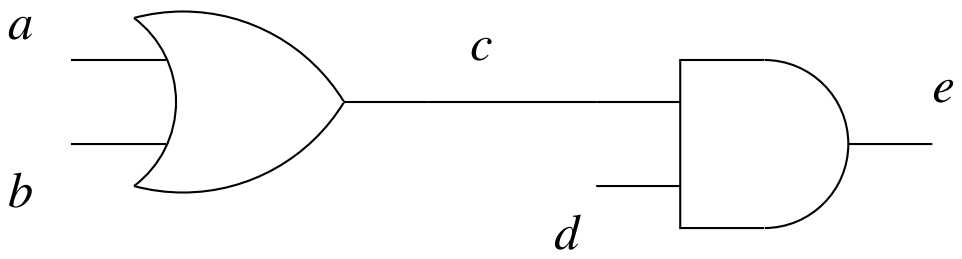
which again is equivalent to the unsatisfiability of

$$f \wedge \bar{l} \wedge \overline{C_i} \qquad \text{for } i = 1 \ldots n$$

This can be done directly on the CNF and needs n unsatisfiability checks.

Variable Instantiation for Tseitin Encodings

$$(\overline{a} \vee c) \qquad (c \vee \overline{e})$$
$$(\overline{b} \vee c) \qquad (d \vee \overline{e})$$
$$(a \vee b \vee \overline{c}) \qquad (\overline{c} \vee \overline{d} \vee e)$$



$$\left.\begin{array}{l} \not\models \; f \wedge \overline{c} \wedge \overline{(a \vee b)} \\ \not\models \; f \wedge \overline{c} \wedge \overline{(\overline{d} \vee e)} \end{array}\right\} \Rightarrow \quad \text{add } c \text{ as unit}$$

requires two satisfiability checks while ATPG for c s-a-1 needs just one run

Stålmarck's Method and Recursive Learning

- orginally Stålmarck's Method works on "sea of triplets" [Stålmarck'89]

  $x = x_1 @ \ldots @ x_n$ with $@$ boolean operator

  - equivalence reasoning + structural hashing + test rule
  - test rule translated to CNF f: $f \Rightarrow (BCP(f \wedge x) \cap BCP(f \wedge \bar{x}))$

    add to f units that are implied by both cases x and $\bar{x}$

- Recursive Learning [KunzPradhan 90ties]

  - originally works on circuit structure
  - idea is to analyze all ways to justify a value, intersection is implied
  - translated to CNF f which contains clause $(l_1 \vee \ldots \vee l_n)$

    BCP on all l_i seperately and add intersection of derived units

Further CNF Simplification Techniques

- failed literals, various forms of equivalence reasoning
- HyperBinaryResolution [BacchusWinter]
  - binary clauses obtained through hyper resolution
  - avoid adding full transitive closure of implication chains
- Variable and Clause Elimination
  - via subsumption and clause distribution, and related techniques

    see our SAT'05 paper and talk by Niklas Éen for further references
  - autarkies and blocked clauses [Kullman]

Summary Circuit based Simplification vs. CNF simplification

- circuit reasoning/simplification can use structure of circuit
  - graph structure (dominators)
  - notion of direction (forward and backward propagation)
  - partial models (some inputs do not need to be assigned)
- CNF simplification does not rely on circuit structure
  - ortogonal: can for instance remove individual clauses
- adapt ideas from circuit reasoning to SAT

  (e.g. avoid multiple SAT checks for redundancy removal in CNF)

QBF for Hardware Verification and Synthesis

- rectification problems (actually a synthesis problem)

  $$\exists p[\forall i[g(i,p) = s(i)]]$$

  with parameters p, inputs i, generic circuit g, and specification s

  QBF solvers only used in [SchollBecker DAC'01] otherwise BDDs

- games, open systems, non-deterministic planning applications?

- model checking

  - termination check as in classical (BDD based) model checking

    (only one alternation)

  - acceleration as in PSPACE completeness for QBF proof

    (at most linear number of alternations in number of state bits n)

Decisions Procedures for Verification using SAT

- specific workshop: Satisfiability modulo Theories (SMT'05)
- examples
  - processor verification [BurchDill CAV'94], [VelevBryant JSC'03]
  - translation validation [PnueliStrichmanSiegel'98]
- eager approach: translate into SAT
- lazy approach
  - augment SAT solver to handle non-propositional constraints
  - in each branch: SAT part satisfiable, check non-propositional theory

Examples for Using SAT in Software Verification

- [JacksonVaziri ISSTA'00] Alloy
  - bounded model checking of OO modelling language Alloy
  - checks properties of symbolic simulations with bounded heap size
- [KroeningClarkeYorav DAC03] CBMC
  - targets equivalence checking of hardware models
  - bounded model checking of C resp. Verilog programs
- [XieAiken POPL'05] Saturn
  - LINT for lock usage in large C programs (latest Linux kernel)
  - neither sound nor complete, but 179 bugs out of 300 warnings

QBF in Software Verification

[CookKröningSharygina – SMC'05]

- model: asynchronous boolean programs
  - parallel version of those used in SLAM, BLAST or MAGIC
- symbolic representation of set of states
  - related work uses BDDs, [CookKröningSharygina] boolean formulas
- termination check for reachability (partially explicit)
  - trivial with BDDs as symbolic representation
  - QBF decision procedure for boolean formulas $\Leftarrow$ QUANTOR
- SAT/QBF version seems to scale much better than BDDs

Summary

- applications fuel interest in SAT/QBF
  - learn from specific techniques ...
  - ... and generalize
- SAT/QBF as core technologies for verification
  - simplified setting in SAT (CNF)
    - on one hand restricts what can be done
    - focus on generic techniques
    - efficient implementations