Circuit versus CNF Reasoning for Equivalence Checking

Armin Biere

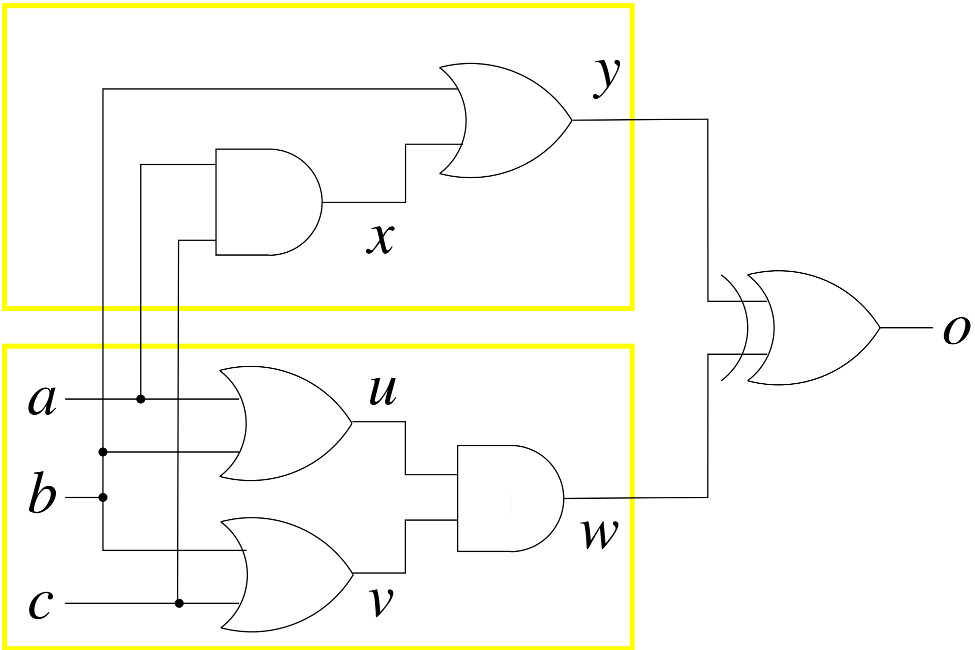
Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Equivalence Checking Workshop 2005

Madonna di Campiglio, Italy
July 29, 2005

Example of Tseitin Transformation: Circuit to CNF

[Tseitin'68]



$$
\begin{array}{r}
o \; \wedge \\
(x \leftrightarrow a \wedge c) \; \wedge \\
(y \leftrightarrow b \vee x) \; \wedge \\
(u \leftrightarrow a \vee b) \; \wedge \\
(v \leftrightarrow b \vee c) \; \wedge \\
(w \leftrightarrow u \wedge v) \; \wedge \\
(o \leftrightarrow y \oplus w)
\end{array}
$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$

Preprocessing SAT

- general idea:
  - simplify CNF before applying complete DPLL algorithm
  - heuristic: simpler CNF is easier to solve
  - metric for simpler: smaller, e.g. less clauses or literals
- for instance failed literal rule
  - *assume* one literal l by setting it to *true*
  - perform boolean constraint propagation (BCP)
  - if BCP generates conflict (empty clause), then permanently add l
  - continue until no more literals are added $\Rightarrow$ saturate

Resolution

[Robinson]

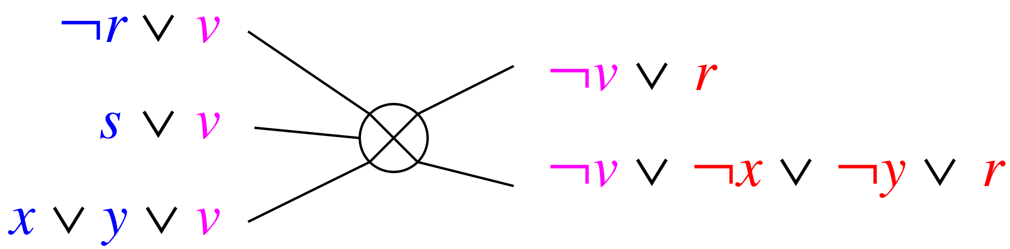
- method to derive logically implied clauses
  - applicable to **resolvees** $(l_1 \vee \ldots \vee l_n \vee v)$ and $(\neg v \vee k_1 \vee \ldots \vee k_m)$
  - with matching variable v
  - implied **resolvent** $(l_1 \vee \ldots \vee l_n \vee k_1 \vee \ldots \vee k_m)$ can be added

- special cases
  - trivial resolvent: $(a \vee b \vee v) \otimes (\bar{v} \vee \bar{a}) \equiv (a \vee b \vee \bar{a}) \equiv 1$
  - unit resolution: $(l_1 \vee \ldots \vee l_n \vee v) \otimes (\bar{v}) \equiv (l_1 \vee \ldots \vee l_n)$
  - resolution of empty clause: $(v) \otimes (\bar{v})$

Elimination of Variables by Resolution (Clause Distribution)

[DavisPutnam'60]

original clauses in which v or $\overline{v}$ occurs:



add non-trivial resolvents:

$(s \vee r)$, $(x \vee y \vee r)$, and $(s \vee \neg x \vee \neg y \vee r)$

remove original clauses

Issues with Clause Distribution

- number of added clauses quadratic in worst case
  - solvers using only clause distribution explode in space
- still useful for preprocessing
  - resolution may generate trivial clause $(a \vee b \vee \overline{a})$
  - or even better units $(a \vee v) \otimes (\neg v \vee a) \equiv a$
  - empirically generates many subsumed clauses

    $(a \vee b)$ subsumes $(a \vee b \vee c)$
  - trivial and subsumed clauses do not have to be added

Subsumption

- backward subsumption
  - new clause being added to CNF **subsumes** clause already in CNF
  - old subsumed clause can be removed after adding new clause
  - search clause in CNF containing **all** literals of new clause
- forward subsumption
  - new clause **is subsumed** by clause already in CNF
  - new clause does not have to be added
  - search clause in CNF made of a **subset** of literals of new clause

Signature based Subsumption Techniques for Propositional CNF

[Biere'04,ÉenBiere'05]

- signature bit for each literal $h(l) \in \{0, \ldots, 31\}$
  - signature of literal is a 32-bit word: $\sigma(l) = 2^{h(l)}$ ($1 << h(l)$ in C)
  - signature of clause is a 32-bit word: $\sigma(C) = \bigcup_{l \in C} \sigma(l)$
  - necessary condition: C subsumes D $\Rightarrow$ $\sigma(C) \subseteq \sigma(D)$
- backward subsumption
  - traverse clauses of a single literal of new clause
  - signature subset check avoids full literal subset check in many cases

Signature based Subsumption Techniques for Propositional CNF cont.

- originally implemented in QBF solver Quantor [Biere'04]
  - fast subsumption essential for resolution based variable elimination
- SAT Preprocessor Satelite [ÉenBiere'05]
  - fast subsumption has similar impact as in QBF
  - SateliteGTI = Satelite + Minisat

    (new version of Minisat by Sörenssen + Éen)
  - SateliteGTI winner of all industrial categories in SAT'05 competition
- forward subsumption: add clauses in reverse order, backward subsume

  (faster way: 1-watched literal scheme [Zhang'05])

More Features in Satelite

- self subsuming resolution:
  - allows to remove *single* literals from clauses
  - beside clause distribution and fw/bw subsumption most effective
  - resolvent subsumes one resolvee: $(a \vee \overline{v} \vee c) \otimes (a \vee v) \equiv (a \vee c)$
- efficient scheduler for clause distribution and self subsumption
- functional substitution of gates (cheaper than clause distribution)
- hyper unary resolution: $(a \vee b \vee c) \otimes (\overline{a} \vee b) \otimes (\overline{c} \vee b) \equiv (b)$

Second Level Signature based Subsumption Techniques
[Biere'04]

- avoids traversing occurrence list in many cases

- signature sum of a literal: $\Sigma(l) = \bigcup\{\sigma(D) \mid D \in CNF \text{ and } l \in D\}$

- necessary condition for new clause D to subsume an old clause:

$$\sigma(C) \subseteq \Sigma(l) \quad \text{for all} \quad l \in C$$

- removing clauses
  - it is sound to keep old signature
  - recalculate *accurate* signature sums after many removals

- technique can be extended to *extract gates* and *hyper unary resolution*

Experiments for Second Level Signatures

	sec	v	v'	red	c	c'	red	l	l'	red	sub	2nd hit	1st miss
1	1.05	9	2	73%	55	21	61%	149	68	54%	28	76.6%	62.0%
2	0.15	2	0	98%	11	0	97%	28	1	96%	8	70.1%	39.6%
3	1.43	14	4	73%	71	33	54%	187	107	43%	30	81.0%	55.1%
4	2.27	28	3	89%	135	25	81%	352	83	76%	95	72.9%	43.7%
5	0.69	9	0	93%	39	4	89%	100	15	84%	30	69.6%	44.1%
6	8.98	51	5	90%	356	87	76%	972	259	73%	1091	31.5%	16.8%
7	0.59	7	0	100%	40	0	100%	102	0	100%	29	68.9%	43.7%
8	6.22	58	13	76%	277	142	49%	714	453	36%	165	72.8%	69.7%
9	7.04	63	15	76%	307	162	47%	794	520	34%	180	72.7%	70.6%
10	6.40	59	9	84%	322	73	77%	850	240	72%	322	64.6%	33.0%
11	2.78	32	7	76%	149	53	64%	393	179	54%	75	80.5%	56.5%
12	3.95	39	11	70%	193	89	54%	511	302	41%	84	80.5%	55.2%
13	1.34	13	3	74%	65	29	55%	172	97	44%	25	79.8%	68.1%

Why Hyper Unary Resolution?

original CNF including clauses modelling an AND gate $a = b \wedge c \wedge d$

$$(\overline{a} \vee b) \wedge (\overline{a} \vee c) \wedge (\overline{a} \vee d) \wedge \underbrace{(a \vee \overline{b} \vee \overline{c} \vee \overline{d})}_{\text{base clause}}$$

new clause $(\overline{b} \vee \overline{c} \vee \overline{d})$

backward subsumes base clause of AND gate and prevents gate extraction

however hyper resolution with the binary side clauses of the AND gate

$$(\overline{a} \vee b) \otimes (\overline{a} \vee c) \otimes (\overline{a} \vee d) \otimes (\overline{b} \vee \overline{c} \vee \overline{d}) \equiv a$$

results in unit clause

similar techniques for other subsumptions of base or side clauses

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)

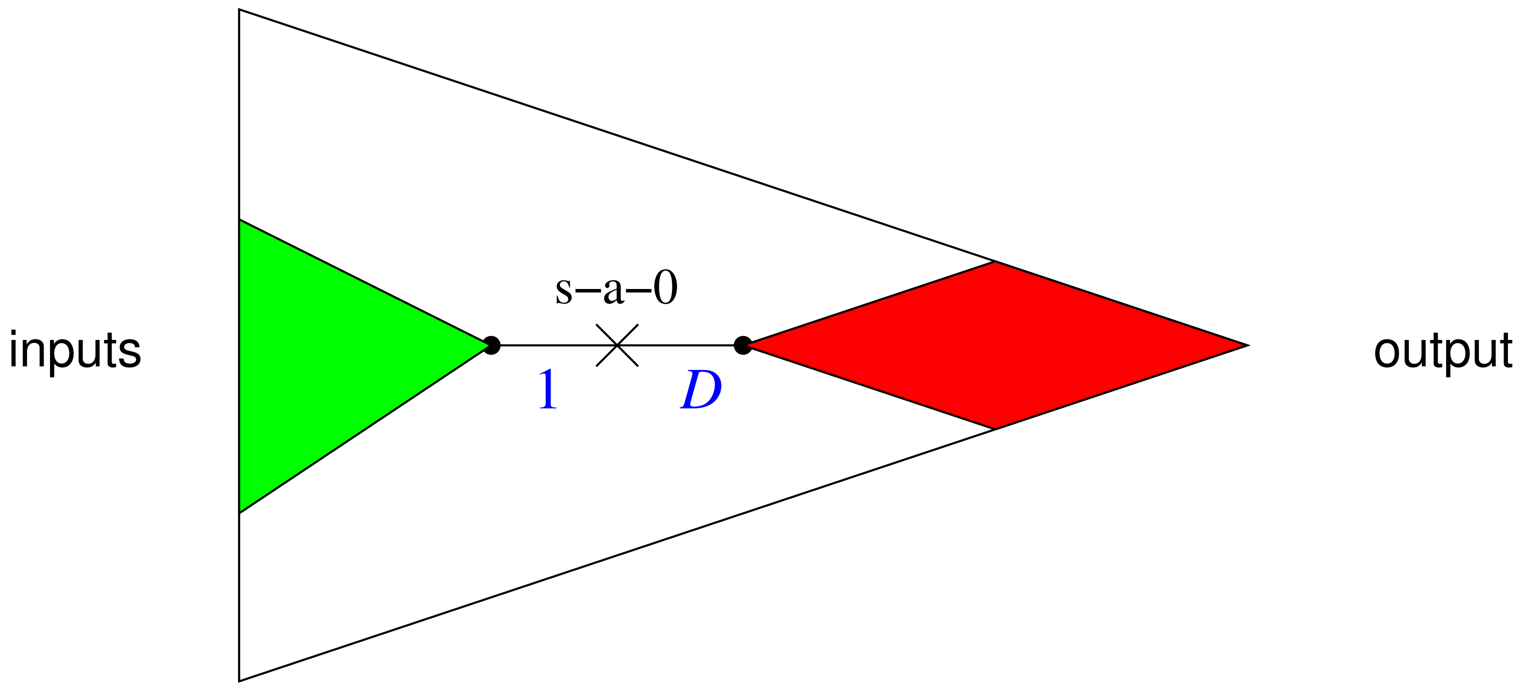
$$1 \wedge D \equiv D \qquad 0 \wedge D \equiv 0 \qquad \overline{D} \wedge D \equiv 0 \qquad \text{etc.}$$

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

Fault Injection for S-A-0 Fault

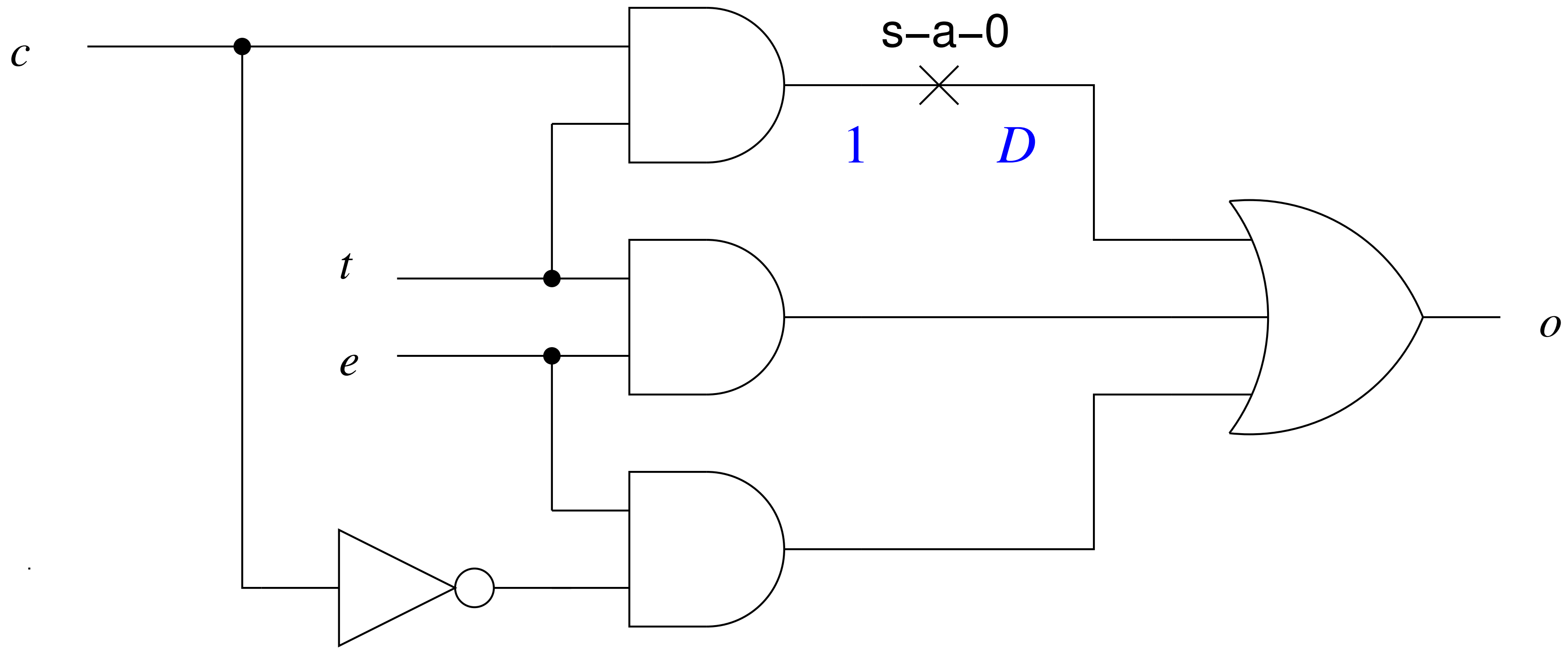
assume opposite value 1 before fault

(both for fault free and faulty circuit)

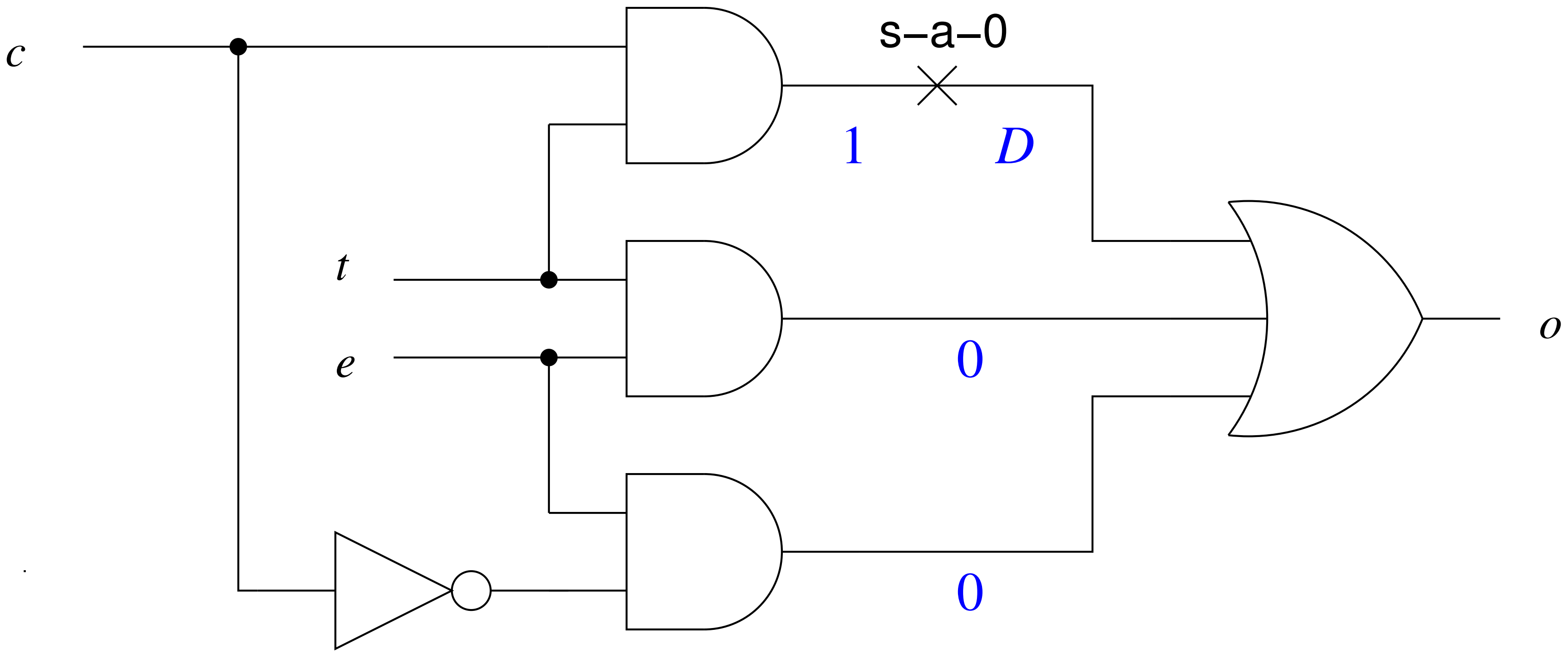


assume difference value D after fault

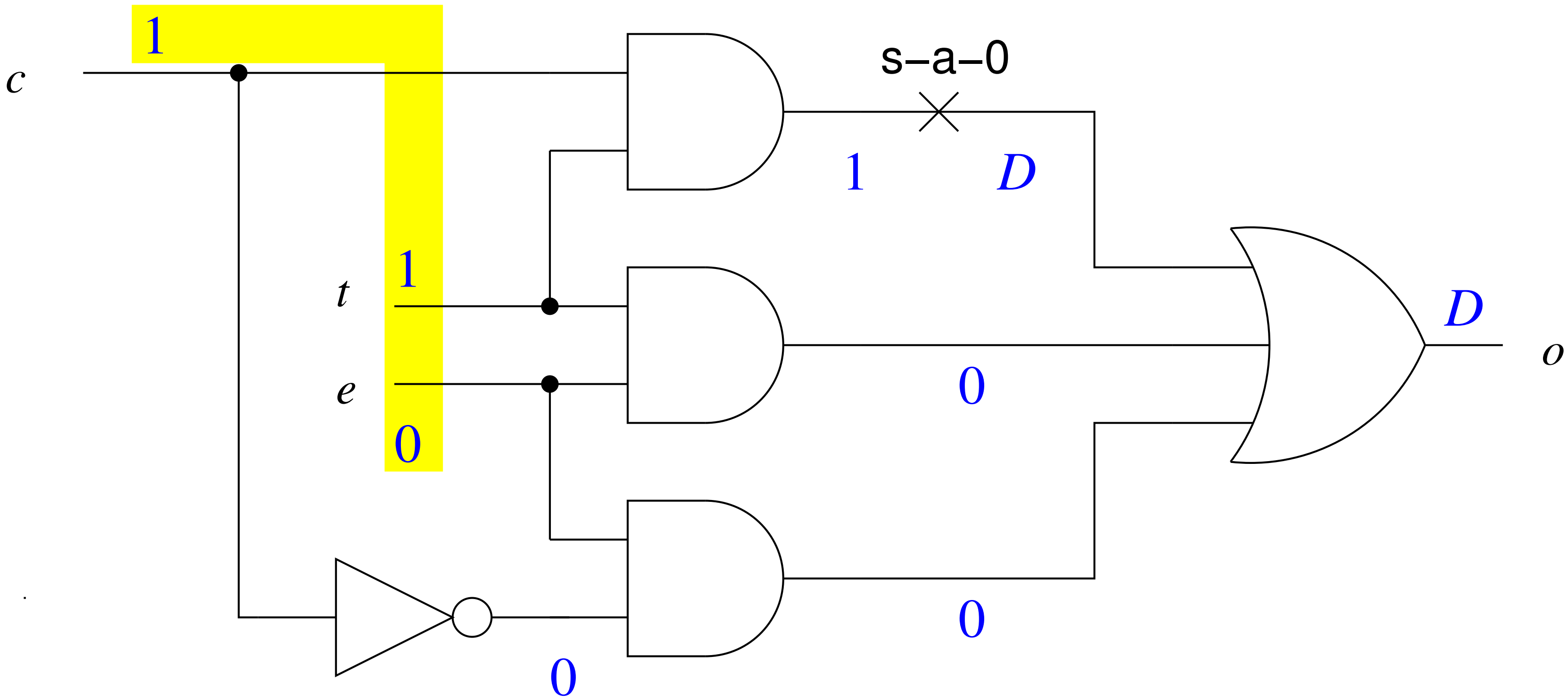
D-Algorithm Example: Fault Injection



D-Algorithm Example: Path Sensitation



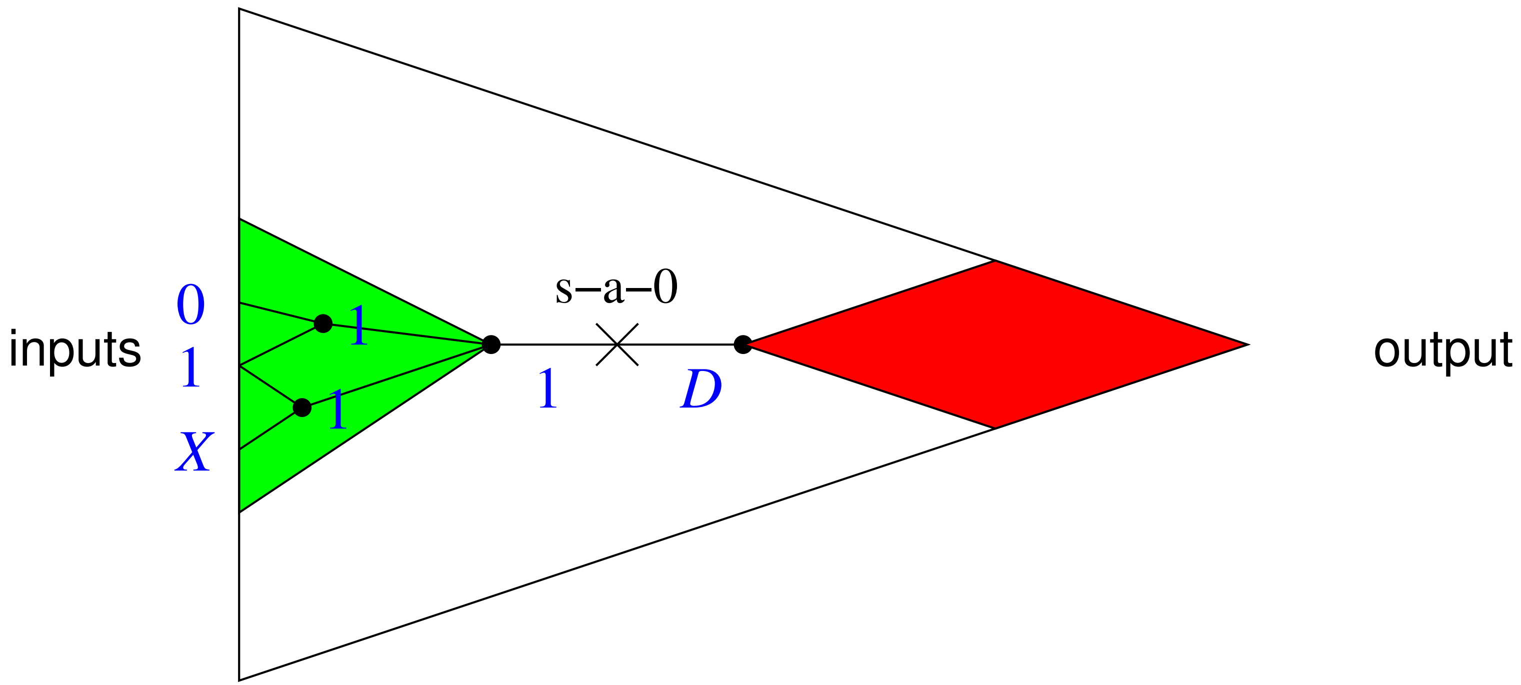
D-Algorithm Example: Propagation



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

Justification

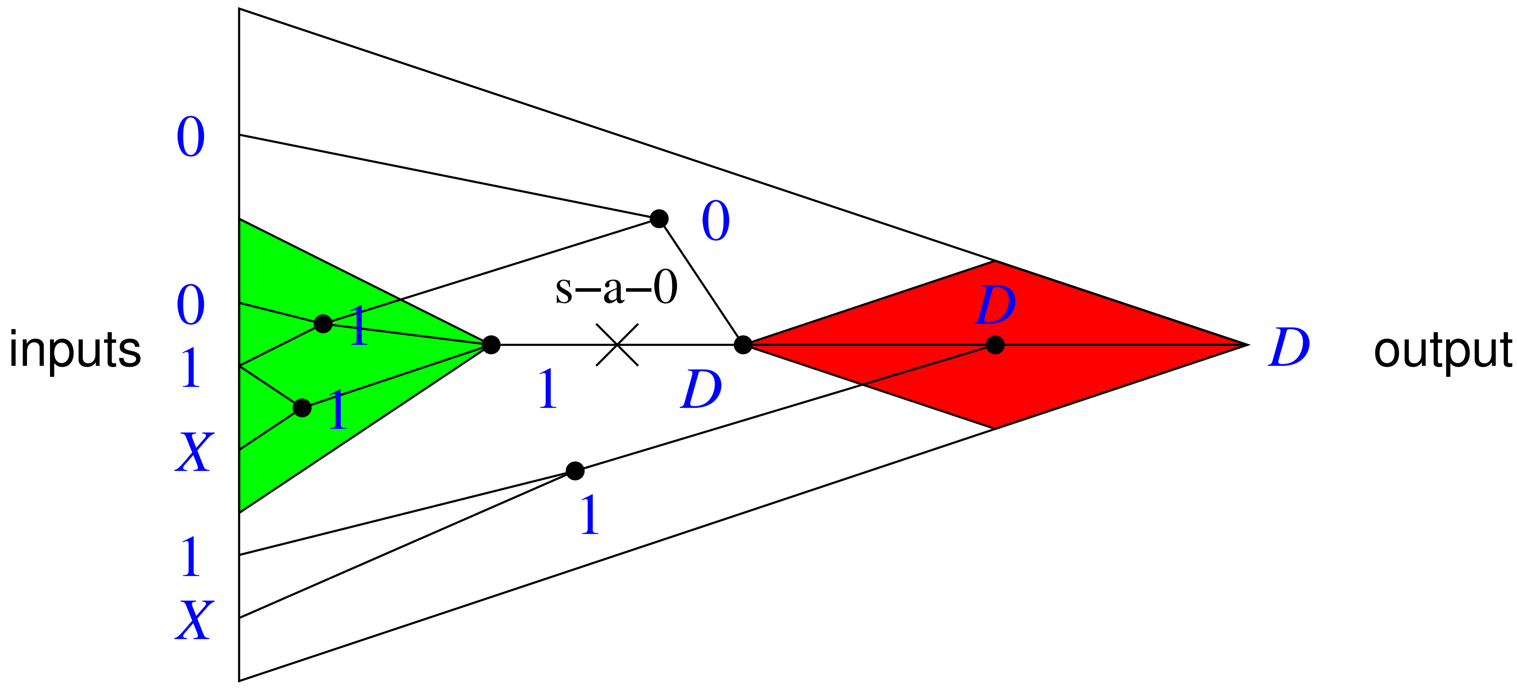
generate partial input vector to justify 1



only backward propagation, remaining unassigned inputs can be arbitrary

Observation

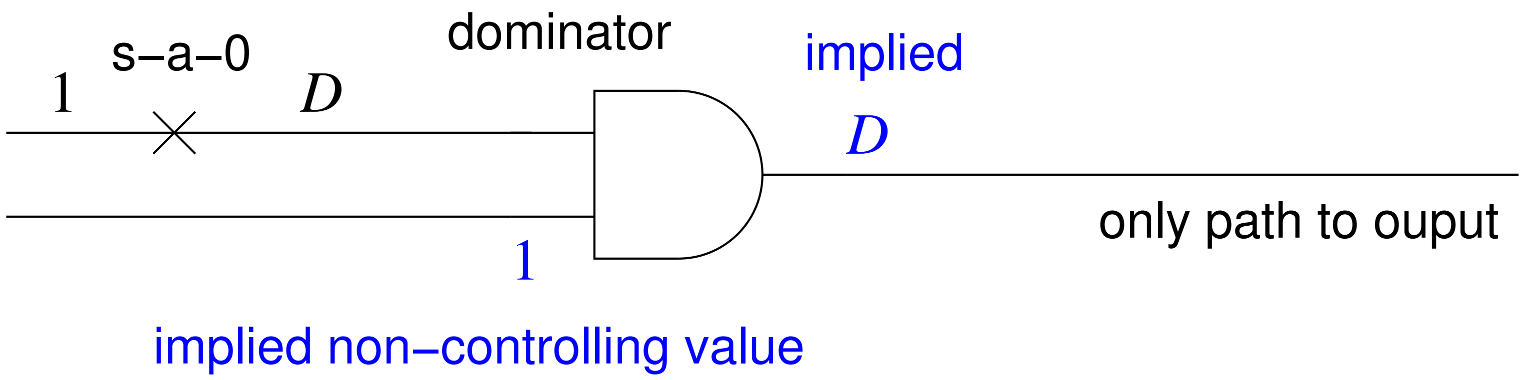
extend partial input vector to propagate D or $\overline{D}$ to output



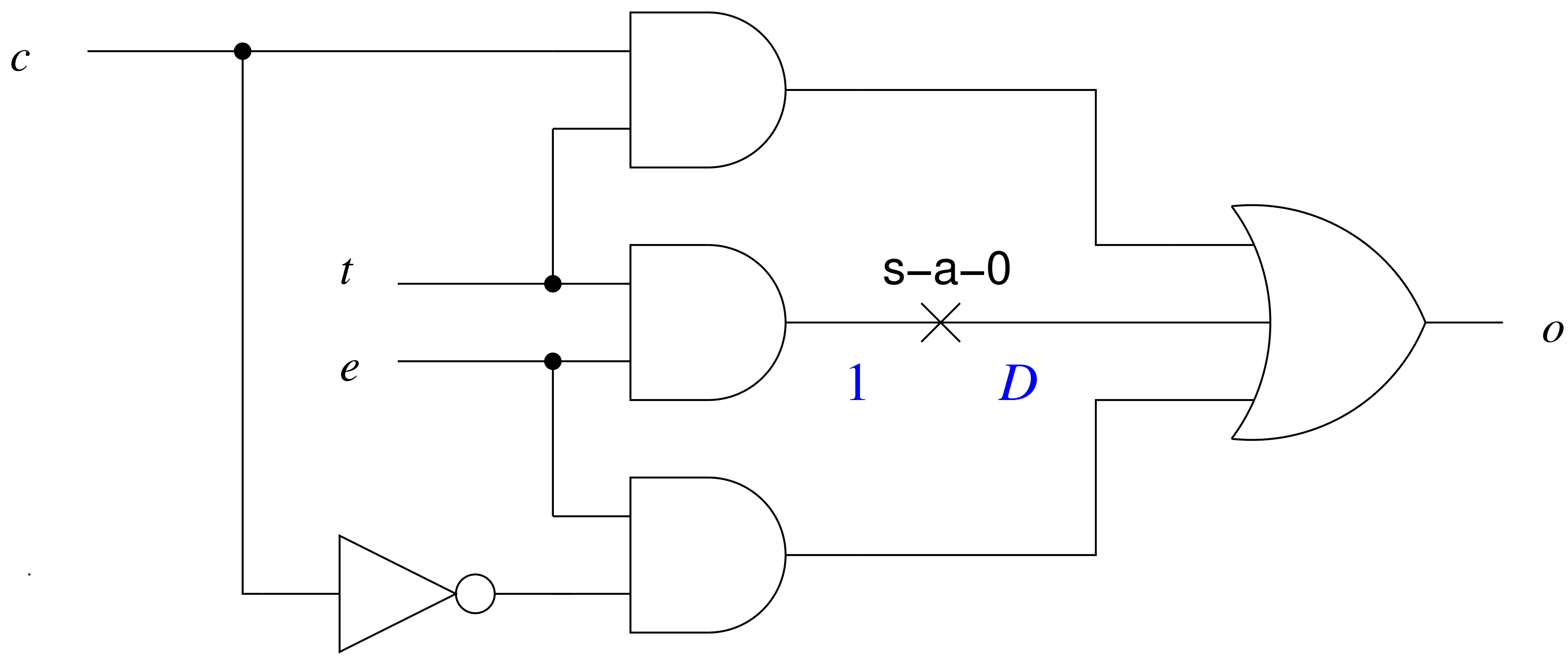
forward propagation of D and $\overline{D}$, backward propagation of 0 and 1

Dominators and Path Sensitation

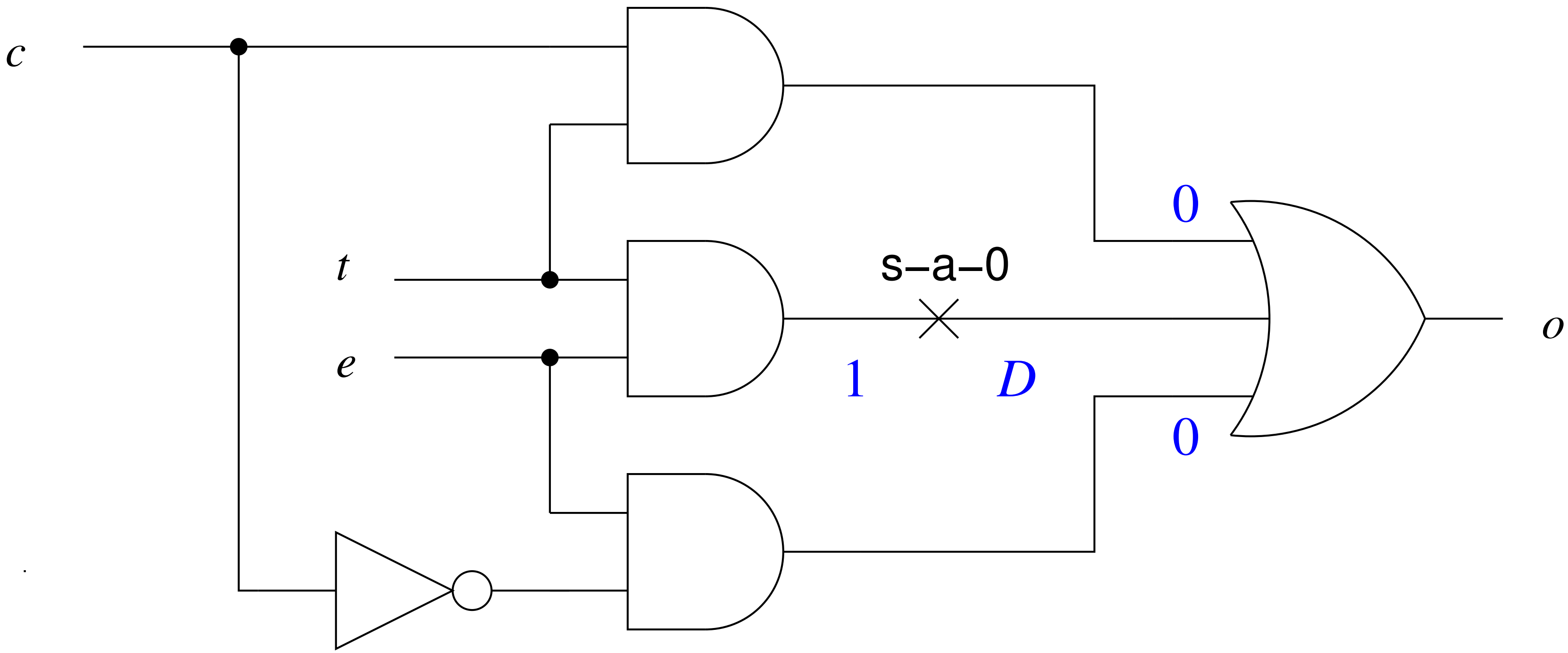
- idea: use circuit topology for additional necessary conditions
  - assign and propagate these conditions after fault injection
- gate dominates fault iff every path from fault to output goes through it
  - more exactly we determine wires (input to gates) that dominate a fault
- if input dominates a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



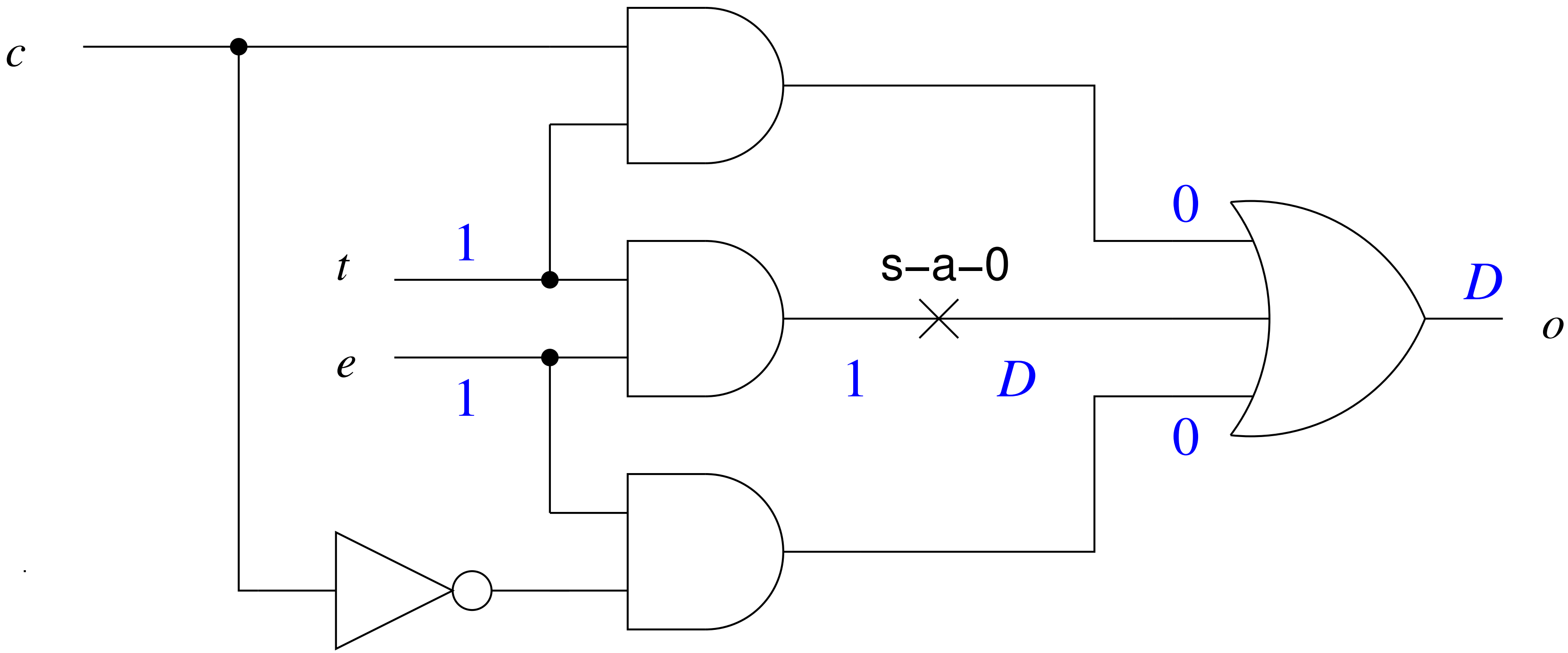
Redundancy Removal with D-Algorithm: Fault Injection



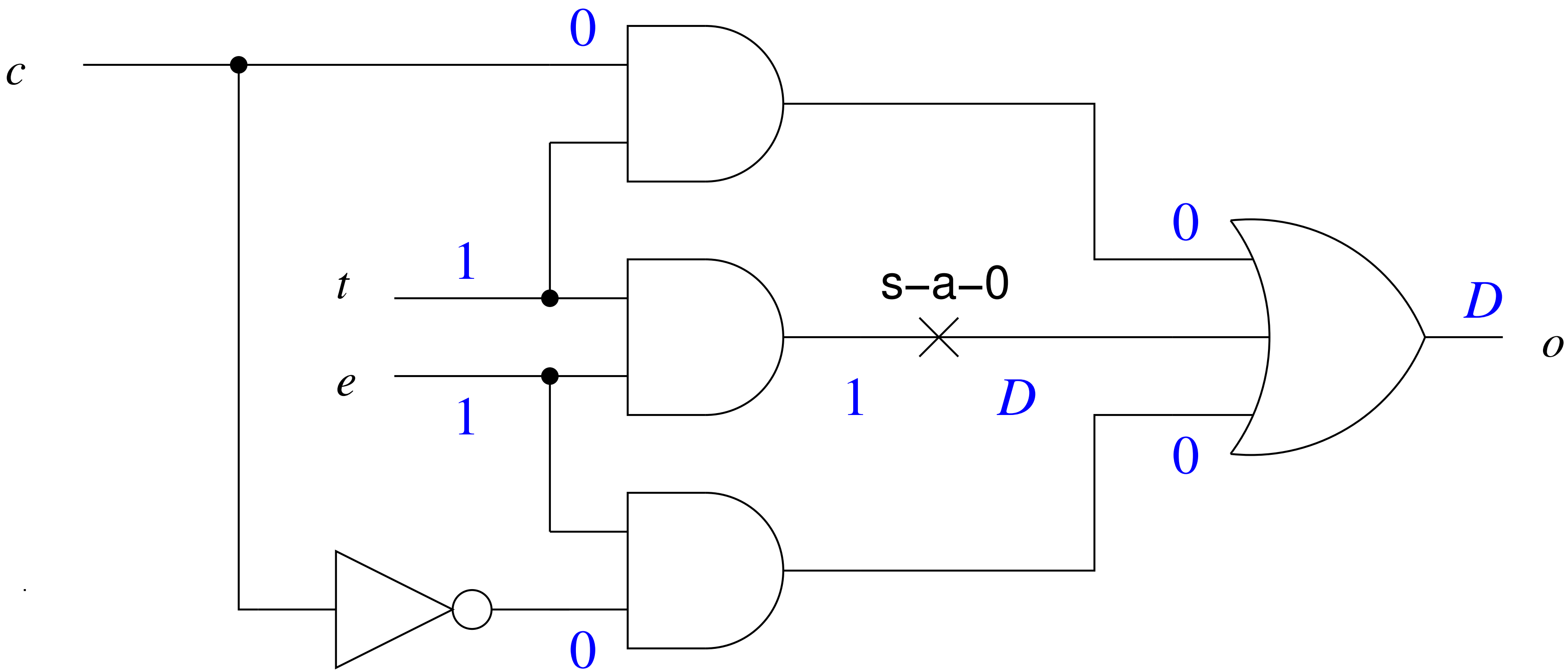
Redundancy Removal with D-Algorithm: Path Sensitation



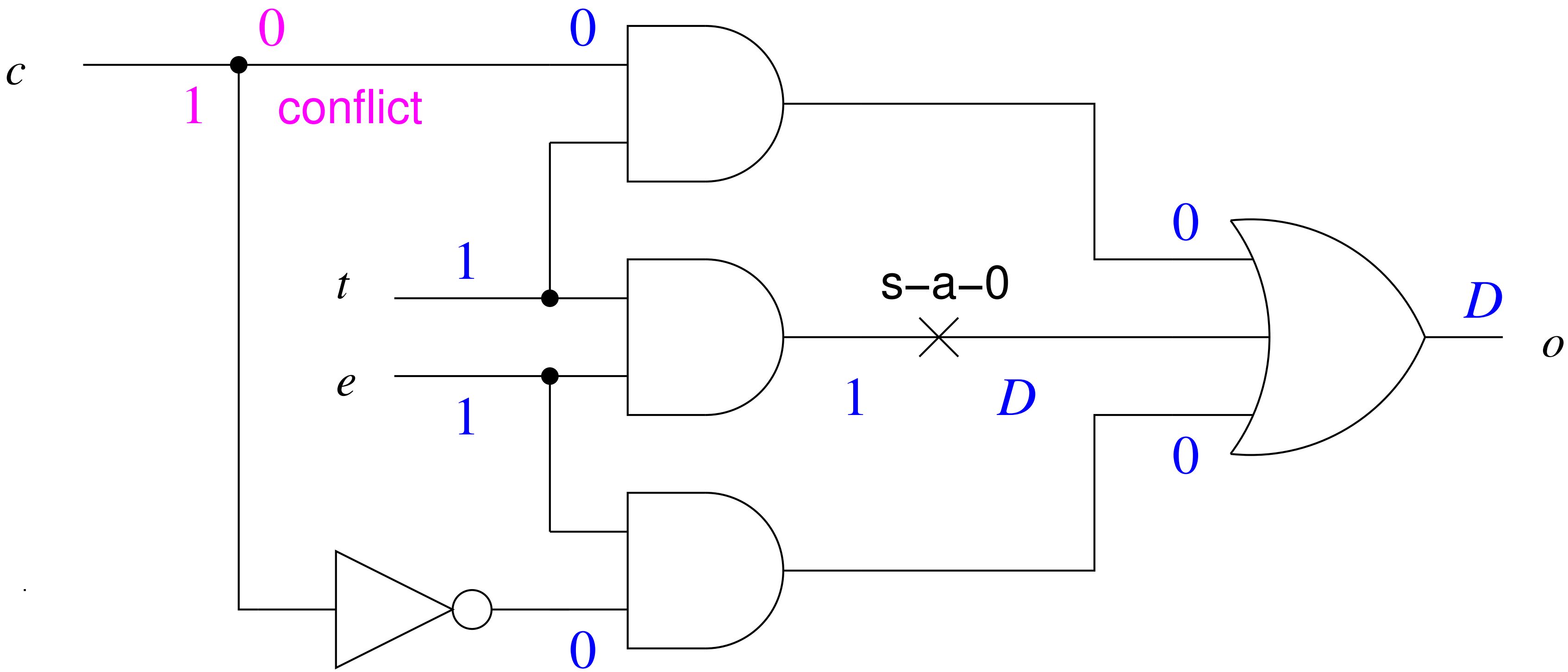
Redundancy Removal with D-Algorithm: 1st Propagation



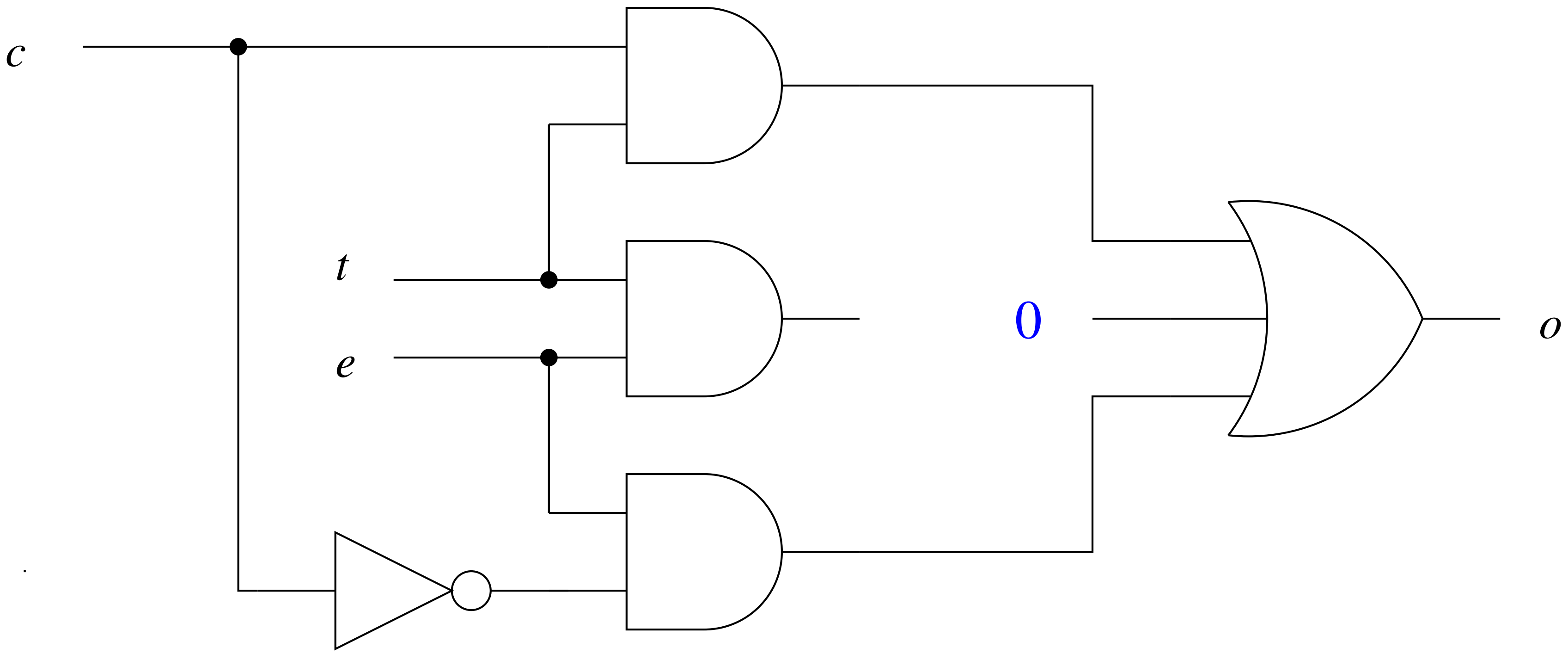
Redundancy Removal with D-Algorithm: 2nd Propagation



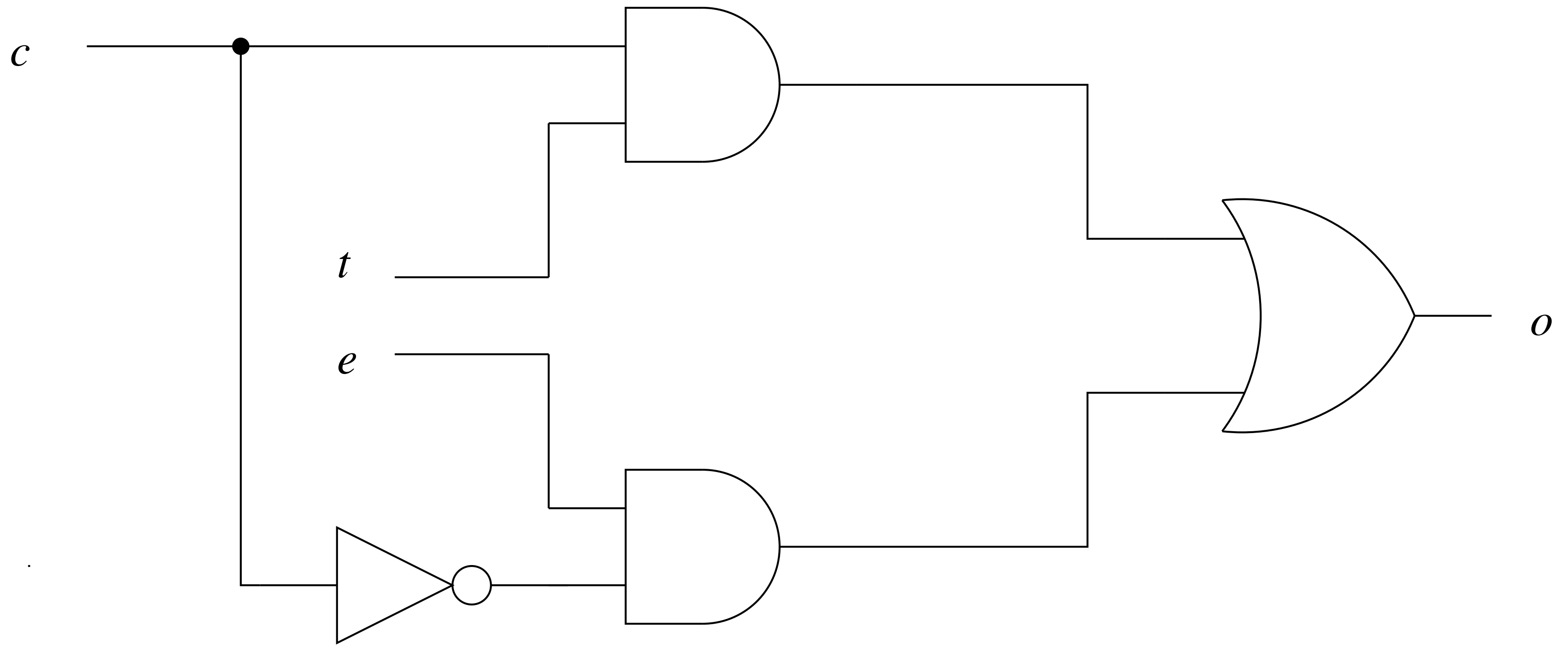
Redundancy Removal with D-Algorithm: Untestable



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

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

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

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

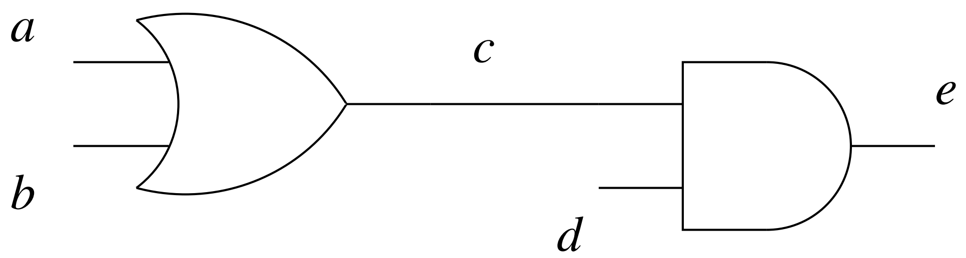
This can be done directly on the CNF and needs n unsatisfiability checks.

Variable Instantiation for Tseitin Encodings

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

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

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



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

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]

  $$x = x_1 @ \dots @ x_n \qquad \text{with } @ \text{ boolean operator}$$

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

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

- 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 $(l_1 \vee \dots \vee l_n)$

    BCP on all l_i separately and add intersection of derived units

Further CNF Simplification Techniques

- failed literals, various forms of equivalence reasoning

- hyper binary resolution [BacchusWinter'03,GershmanStrichman'05]
  - add binary clauses obtained through hyper resolution
  - avoid adding full transitive closure of implication chains
  - equivalence reasoning through SCC detection in binary clause graph
  - as Stålmarck's procedure subsumes structural hashing

- variable and clause elimination
  - autarkies and blocked clauses [Kullman]

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
  - orthogonal: can for instance remove individual clauses

- adapt ideas from circuit reasoning to SAT

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