COMP0009 Logic — Problem Sheet 7: SAT Solving

Part A — Finding a counter-model

Alice, Bob and Charlie share a network on which nobody trusts anybody, and Mallory attacks it. The network has a *rulebook*: lines that say which settings of its switches are allowed, who may read which file, who holds which key. Charlie's job is to answer questions of the form "does this follow from the rulebook?", that is, is it true in every world that obeys the rules? Until now his answers have been *proofs*, trees of inference steps, and a proof is hard to find: at every step many rules could apply, and nothing on the page says which. This week he turns the search round. He treats "does it follow?" as he would a bug: suppose a counterexample, write down what the rules force, and either the page collapses, which is the proof, or a world is left on it, which is the bug report. This is licensed by a fact the notes prove, which you may take on trust: a claim has no proof from the rules exactly when some world obeys the rules and breaks the claim, and such a world gives a *counter-model*. So a search for such a world is as good as a proof.

In the four scenes below the search becomes a machine. A rule becomes a list of *forbidden patterns* of switches; the search on switches is a procedure called DPLL; a "someone" in a rule is given a name once and for all; and the search for a world that obeys rules about *everyone* runs level by level through the things the rules can name. You will run the machine by hand, check its answers, catch Mallory's forgeries, and meet a question on which the machine runs for ever.

Marks are in square brackets; the whole of Part A is worth 25.

A1. Clauses.

Every node of Charlie's network has configuration switches, each on or off, and the rulebook says which settings are allowed. Four switches matter this week: `relay` (the node passes packets on for others), `verify` (it checks the signature on every packet it handles), `cache` (it keeps copies of recent packets) and `offline` (it serves from its cache when the network is down). We write them as the letters p, q, r and s: p says "`relay` is on", and $\neg p$ says it is off. A *setting* of the switches is a valuation v, true or false for each letter.

Five lines of the rulebook concern these four switches.

1. A relaying node verifies what it relays.
2. A relaying node keeps a cache.
3. Every node relays or works offline; with neither it is useless.
4. An offline node serves from its cache.
5. Verifying and caching do not mix: the verifier rejects any packet it did not see arrive.

Every line forbids some patterns of the switches and allows the rest. Line 1 is broken by exactly one pattern of p and q: `relay` on, `verify` off. So it says no more than "never p on and q off", and the plainest way to write that is $\neg p \vee q$: one literal for each switch in the forbidden pattern, each saying the opposite of what the pattern says.

- A *literal* ℓ is a letter p or a negated letter $\neg p$. Its complement $\bar{\ell}$ says the opposite: $\bar{p}$ is $\neg p$ and $\overline{\neg p}$ is p.
- A *clause* C is a disjunction $\ell_1 \vee \cdots \vee \ell_n$ of literals, with $n \geq 0$, read as "at least one of these". The order and repetition of literals do not matter, so we also treat

a clause as the set $\{\ell_1, \dots, \ell_n\}$ of its literals. The *empty clause*, with no literals, is written $\square$; it demands at least one of nothing, so no setting satisfies it. A clause with one literal is a *unit clause*: it fixes a switch.

- A *clause set* $\mathcal{S}$ is a finite set of clauses. A valuation v *satisfies* a clause, $v \vDash C$, if it makes some literal of C true, and satisfies $\mathcal{S}$ if it satisfies every clause in it; $\mathcal{S}$ is *satisfiable* if some valuation does.

A clause whose literals are on different letters forbids exactly one pattern of those switches, and a clause set forbids a list of patterns. A setting satisfies the set exactly when it avoids every pattern on the list.

A constraint that is not written as a clause is turned into clauses in two steps. First the translation nnf removes every arrow and pushes every negation down to the letters; here $\neg$ is a connective in its own right, and $\top$ is short for $\neg\bot$. Each equation is a law of truth tables (De Morgan's laws, double negation, the arrow as $\neg\psi \vee \chi$):

$$\begin{aligned}
&\mathrm{nnf}(\ell) = \ell \text{ for a literal, } \bot \text{ or } \top, && \mathrm{nnf}(\neg\neg\psi) = \mathrm{nnf}(\psi),\\
&\mathrm{nnf}(\psi \wedge \chi) = \mathrm{nnf}(\psi) \wedge \mathrm{nnf}(\chi), && \mathrm{nnf}(\neg(\psi \wedge \chi)) = \mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\neg\chi),\\
&\mathrm{nnf}(\psi \vee \chi) = \mathrm{nnf}(\psi) \vee \mathrm{nnf}(\chi), && \mathrm{nnf}(\neg(\psi \vee \chi)) = \mathrm{nnf}(\neg\psi) \wedge \mathrm{nnf}(\neg\chi),\\
&\mathrm{nnf}(\psi \to \chi) = \mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\chi), && \mathrm{nnf}(\neg(\psi \to \chi)) = \mathrm{nnf}(\psi) \wedge \mathrm{nnf}(\neg\chi).
\end{aligned}$$

The result, $\mathrm{nnf}(\varphi)$, is in *negation normal form*: literals, $\bot$ and $\top$ joined by $\wedge$ and $\vee$ only. Second, the translation cnf distributes every $\vee$ over every $\wedge$, which leaves a conjunction of clauses, and the *clause form* $\mathrm{cl}(\varphi)$ is the clause set of $\mathrm{cnf}(\mathrm{nnf}(\varphi))$. Read as clause sets, the equations of cnf say: a literal ℓ gives the unit clause $\{\ell\}$; $\bot$ gives $\{\square\}$ and $\top$ gives the empty clause set; for formulas ν_1, ν_2 in negation normal form, $\nu_1 \wedge \nu_2$ pools the clauses of both sides; and $\nu_1 \vee \nu_2$ joins every clause of one side to every clause of the other, $C \cup D$ for C from the left and D from the right. The clause form is satisfied by exactly the settings that make the constraint true.

Example. Line 1 says "if `relay` is on then `verify` is on": $p \to q$. By the equation for $\to$, $\mathrm{nnf}(p \to q) = \mathrm{nnf}(\neg p) \vee \mathrm{nnf}(q) = \neg p \vee q$, and $\mathrm{cl}(\neg p \vee q)$ joins the unit clause $\{\neg p\}$ to the unit clause $\{q\}$: the one clause $C_1 = \neg p \vee q$, forbidding p on, q off.

Now a line with a negation over it, "it is not the case that `relay` is on with `cache` off": $\neg(p \wedge \neg r)$. The equation for $\neg(\psi \wedge \chi)$ gives $\mathrm{nnf}(\neg p) \vee \mathrm{nnf}(\neg\neg r)$, and double negation gives $\neg p \vee r$. The clause form is the one clause $\neg p \vee r$, which is line 2.

A search by hand writes down what it has settled so far and reads each constraint against the page. The page is a *partial valuation* u: it gives some letters a value and leaves the rest undecided. We name it by listing the literals it makes true, in braces: $\{p, \neg q\}$ has p on, q off and every other letter undecided, and $\emptyset$ decides nothing. A literal is *true under* u if it is in u, *false* if its complement is, and *undecided* otherwise. Under u, a clause is

- *satisfied* if some literal of it is true under u: the page already obeys it;
- *falsified* if every literal of it is false under u: the page already breaks it;
- a *unit* if it is not satisfied, exactly one literal is undecided and the rest are false: the page leaves it one way out, and that literal must be made true.

Unit propagation is the forced move applied as long as there is one: find a clause that is a unit under u, add its undecided literal to u, and look again. Under $u = \{p, \neg q\}$ the clause $q \vee r$ is a unit, $\neg p \vee q$ is falsified, and $p \vee s$ is satisfied.

(a) Here are the five lines as clauses, with the pattern each forbids. Four entries are missing; fill them in. [1]

$C_1 = \neg p \vee q$	never	p on, q off;
$C_2 = \neg p \vee r$	never	☐
$C_3 =$ ☐	never	p off, s off;
$C_4 = \neg s \vee r$	never	☐
$C_5 =$ ☐	never	q on, r on.

(b) Bob had written lines 1 and 2 as one line, "a relaying node verifies and caches", and line 5 with a negation over it:

(i) $p \to (q \wedge r)$ (ii) $\neg(q \wedge r)$.

Compute $\mathrm{nnf}(\varphi)$ and the clause form of each, showing the equations of the translation you use, and check that you get the clauses of (a). [2]

Hint: in (ii), use the equation for a negation over $\wedge$, as in the example.

(c) Mallory proposes a line of her own: "either the node relays and verifies, or it works offline from its cache", $(p \wedge q) \vee (s \wedge r)$. She says that one line of the rulebook is one clause. Compute the clause form, say how many clauses it has, and beside each clause write the pattern it forbids. [1]

(d) Alice wants her node to relay. Charlie starts a page with $u = \{p\}$.

(i) Under u, say for each of $C_1, \ldots, C_5$ whether it is satisfied, falsified, a unit, or none of these. [1]

(ii) Run unit propagation from u, writing each forced literal with the number of the clause that forced it, until nothing more is forced or a clause is falsified. What does Charlie tell Alice? [1]

A2. DPLL.

Propagation stops when no clause is a unit, and then a guess is needed: choose an undecided switch, try it one way, and if that leads to a falsified clause, back up and try it the other way. Every setting has the switch one way or the other, so the two tries between them cover every setting the page could still become. The search is drawn as a *DPLL tree*. The root carries no literal. The rule *decide* gives the end of a branch two children, labelled p and $\neg p$ for a letter the branch has not yet decided; the rule *unit* gives it one child, the undecided literal of a clause that is a unit on the branch. Both rules apply only to a branch on which no clause is falsified. The *partial valuation of a branch* is the set of literals on it. A branch is *closed* if some clause is falsified on it, *finished* if every clause is satisfied on it, and *complete* if it is one or the other; the tree is *closed* if every branch is.

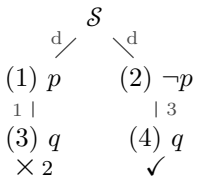
In the drawings, the root is marked $\mathcal{S}$, an edge labelled d is a decision, an edge labelled with a number is a forced move by that clause, a cross with a number marks a closed branch and the clause it falsifies (the rule *close*), and a tick marks a finished branch (the rule *finish*). The procedure below fixes the order of the rules that the definition of a tree leaves open. In the code, u is a partial valuation written as its set of literals, and `u + l` is $u + \ell$, that is, $u \cup \{\ell\}$:

```
procedure DPLL(S)
    return SEARCH(S, {})                    -- u decides nothing yet

procedure SEARCH(S, u)
```

```
    u := PROPAGATE(S, u)
    if some clause of S is falsified under u: return CLOSED
    if every clause of S is satisfied under u: return OPEN(u)   -- a setting
    p := a letter that u does not decide, from a clause of S not satisfied under u
    r := SEARCH(S, u + p)
    if r is OPEN(u'): return r
    return SEARCH(S, u + not p)

procedure PROPAGATE(S, u)
    while no clause of S is falsified under u, and some clause C of S
          is a unit under u, with undecided literal l:
        u := u + l
    return u
```

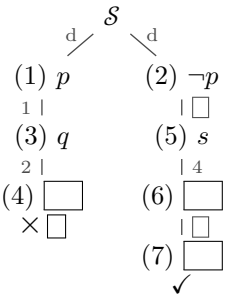
So it propagates until nothing is forced, gives up on a branch as soon as a clause is falsified, stops at the first finished branch, and tries the "on" side of a decision before the "off" side. On this sheet, `PROPAGATE` takes the first unit in the numbering of the clauses, and a decision is on the first letter in the order given that is undecided and occurs in a clause not yet satisfied. Two facts about it, which you may use: it always stops; and it returns `CLOSED` exactly when $\mathcal{S}$ is unsatisfiable, so a closed tree is a certificate that no setting satisfies the clauses, and a finished branch is a setting that does.

Example. Three clauses, $C_1 = \neg p \vee q$, $C_2 = \neg q \vee \neg p$ and $C_3 = p \vee q$. Nothing is a unit under the empty page, so the procedure decides p, on first.



Under p, C_1 forces q (node 3), and then C_2 is falsified: closed. Under $\neg p$, C_3 forces q (node 4), and now C_1 and C_2 are satisfied by $\neg p$ and C_3 by q: finished. The procedure returns `OPEN` with $\{\neg p, q\}$, and anyone can check that setting against the three clauses without knowing how it was found.

(a) Here is the procedure's tree for the five clauses $C_1, \dots, C_5$ of A1, deciding p first. Four node labels and two edge labels are missing. Fill them in, say which branch is closed and by which clause, and which is finished. [2]



Hint: the left branch is the propagation of A1(d).

(b) The finished branch is a setting. Write it out as on/off for the four switches, and check it against each of the five clauses, one line per clause. Was `verify` decided, or forced? [1]

(c) Mallory writes to Alice: "Charlie is wrong. My solver found a setting with `relay` on: `relay` on, `verify` on, `cache` off, `offline` on." Check her setting against the five clauses. Then say why no solver, hers or anyone's, can find a setting with `relay` on, pointing at the part of the tree in (a) that certifies it. [1]

(d) Bob rewrites line 5: verifying and caching may mix after all, if the node also runs `audit`, a fifth switch, which we write t. Auditing is slow, so an auditing node may not relay. The new rulebook is

$$C_1 = \neg p \vee q, \qquad C_2 = \neg p \vee r, \qquad C_3 = p \vee s,$$
$$C_4 = \neg s \vee r, \qquad C_5 = \neg q \vee \neg r \vee t, \qquad C_6 = \neg p \vee \neg t.$$

Draw the procedure's tree, deciding letters in the order p, q, r, s, t, on side first, and stopping at the first finished branch. Read off the setting it returns. Can Alice's node relay now? [2]

Hint: the first decision is p again. On the $\neg p$ side, after propagation, a second decision is needed.

(e) Charlie decided `relay` first because Alice had asked about it. Draw the tree for the five clauses of A1 when `cache` is decided first, on side first. Compare it with the tree of (a): what has changed, and what has not? [1]

A3. Skolem functions and ground instances.

The part of the rulebook with names and quantifiers in it is the *access policy*, which says who may read which file. The relation symbols are $\mathsf{adm}(x)$ ("x is an administrator"), $\mathsf{own}(x, y)$ ("x owns y") and $\mathsf{read}(x, y)$ ("x may read y"), the names are alice, bob, $\mathsf{mallory}$, notes, junk and vault, and the policy has two lines,

(pol1) $\forall x \, \forall y \, (\mathsf{adm}(x) \to \mathsf{read}(x, y))$ administrators may read anything;
(pol3) $\forall x \, \forall y \, (\mathsf{own}(x, y) \to \mathsf{read}(x, y))$ owners may read what they own.

A set of sentences Γ *entails* a sentence φ, written $\Gamma \models \varphi$, when φ is true in every structure in which every member of Γ is true. The fact Charlie relies on is that $\Gamma \models \varphi$ holds exactly when Γ together with $\neg\varphi$ has no model: a model of $\Gamma \cup \{\neg\varphi\}$

is exactly a counter-model. So the search tries to build a model of $\Gamma \cup \{\neg\varphi\}$, and if every attempt collapses, that is the proof. The notes make the search mechanical in three moves.

1. *Clause form* (the first two moves). An algorithm, which the notes treat as one black box (the theorem on the clause form), turns Γ and $\neg\varphi$ into a set of clauses with variables in them. Each "someone" is given a name: a new constant (a *Skolem constant*) or, when the someone depends on who is asking, a new function of the variables it depends on (a *Skolem function*). A variable in a clause means "for everyone". The clause set has a model exactly when $\Gamma \cup \{\neg\varphi\}$ has one. On this sheet the clause forms are given.
2. *Instances* (the third move). The *Herbrand universe* H of the clause set is the set of closed terms built from its constants (one new constant is added if it has none) by its function symbols. H_k is the set of terms of *level* at most k, where a constant has level 0 and $f(t_1, \ldots, t_n)$ has level one more than the largest level of its arguments. The *ground instances* Gr_k are the clauses obtained by putting terms of H_k for the variables of each clause, in every way. Run `DPLL` on Gr_0, then Gr_1, and so on, with each atomic sentence as a letter.

Herbrand's theorem says this is right: the clause set is unsatisfiable exactly when some Gr_k is. By the theorem on the clause form, that is exactly when $\Gamma \models \varphi$. A closed tree at some level is the proof; an `OPEN` answer at a level gives a valuation that satisfies that level's instances and nothing more.

Example. Alice is an administrator, $\mathsf{adm}(\mathsf{alice})$. Does it follow from (pol1) that she may read every file, $\forall y\, \mathsf{read}(\mathsf{alice}, y)$? The negated claim says that there is a file Alice cannot read, and its clause form names that file by a Skolem constant a: the one clause $\neg\mathsf{read}(\mathsf{alice}, a)$. The policy (pol1) has no "someone" to name, and its clause form is $\neg\mathsf{adm}(x) \vee \mathsf{read}(x, y)$; the fact is a unit clause. The clause set:

$$C_1 = \neg\mathsf{adm}(x) \vee \mathsf{read}(x, y), \qquad C_2 = \mathsf{adm}(\mathsf{alice}), \qquad C_3 = \neg\mathsf{read}(\mathsf{alice}, a).$$

Its constants are a and alice and it has no function symbol, so $\mathrm{H} = \mathrm{H}_0 = \{a, \mathsf{alice}\}$. The ground instances at level 0 are C_1 at the four pairs of names, in the order (a, a), (a, alice), (alice, a), $(\mathsf{alice}, \mathsf{alice})$, numbered 1 to 4, then C_2 and C_3 as 5 and 6. The procedure's tree:

$$\begin{array}{c} \mathcal{S} \\ {\scriptstyle 5\ |} \\ (1)\ \mathsf{adm}(\mathsf{alice}) \\ {\scriptstyle 3\ |} \\ (2)\ \mathsf{read}(\mathsf{alice}, a) \\ \times 6 \end{array}$$

The unit 5 forces $\mathsf{adm}(\mathsf{alice})$; then instance 3, $\neg\mathsf{adm}(\mathsf{alice}) \vee \mathsf{read}(\mathsf{alice}, a)$, forces $\mathsf{read}(\mathsf{alice}, a)$; and the unit 6 is falsified. Closed at level 0 with no decision: the policy entails the claim. Read as a search: suppose there is a file Alice cannot read, call it a; she is an administrator, so she may read a; but a was the file she cannot read.

(a) Here are three claims, each with the clause form of its negation. Say in words what each Skolem symbol stands for. [2]

(i) If someone owns the vault, then someone may read the vault,

$$(\exists x\, \mathsf{own}(x, \mathsf{vault})) \to \exists x\, \mathsf{read}(x, \mathsf{vault});$$

the clauses $\mathsf{own}(c, \mathsf{vault})$ and $\neg\mathsf{read}(z, \mathsf{vault})$.

(ii) Everyone may read something, $\forall x\, \exists y\, \mathsf{read}(x, y)$: the clause $\neg\mathsf{read}(b, y)$.

(iii) Someone may read everything, $\exists x\, \forall y\, \mathsf{read}(x, y)$: the clause $\neg\mathsf{read}(x, g(x))$.

Why does the negation of (i) need a Skolem symbol in one clause and not in the other? And why does (ii) need a constant where (iii) needs a function?

Hint: read each negation aloud first, with "someone" and "nobody".

(b) The policy used to have a different second line, (pol2) $\forall x\, \forall y\, (\mathsf{own}(x, y) \to \mathsf{adm}(x))$, "anyone who owns something is an administrator", and (pol3) replaced it. Mallory owns a file of her own, $\mathsf{own}(\mathsf{mallory}, \mathsf{junk})$. An auditor asks Charlie whether the old policy, (pol1) and (pol2), together with that fact, entails that someone may read the vault: $\exists x\, \mathsf{read}(x, \mathsf{vault})$. Charlie answers by machine.

(i) The clause form of the two policy lines, the fact and the negated claim is

$C_1 = \neg\mathsf{adm}(x) \vee \mathsf{read}(x, y)$, $\quad C_2 = \neg\mathsf{own}(x, y) \vee \mathsf{adm}(x)$,

$C_3 = \mathsf{own}(\mathsf{mallory}, \mathsf{junk})$, $\quad C_4 = \neg\mathsf{read}(x, \mathsf{vault})$.

Why does the negated claim need no Skolem symbol? Write down H_0, count the ground instances at level 0, and say why level 0 is all there is here. [1]

(ii) Run the search on them: write each forced move with the instance that forces it, and the instance that is falsified. (Instances of a rule you never use need not be written out.) [1]

Read the closed tree as a search in words. Whom does it name, and what chose the name?

(c) Bob adds a line, "everyone owns something": (own) $\forall x\, \exists y\, \mathsf{own}(x, y)$. Charlie asks whether (pol3) and (own) entail that everyone may read something, $\forall x\, \exists y\, \mathsf{read}(x, y)$, the claim of (a)(ii).

(i) The clause form of (own) is $\mathsf{own}(x, f(x))$, with a Skolem function f, and the negated claim is the clause of (a)(ii), so the clause set $\mathcal{T}$ of the three sentences is

$C_1 = \neg\mathsf{own}(x, y) \vee \mathsf{read}(x, y)$, $\quad C_2 = \mathsf{own}(x, f(x))$, $\quad C_3 = \neg\mathsf{read}(b, y)$.

What does $f(x)$ stand for? Write down H_0, H_1 and H_2. What does $f(f(b))$ stand for? [1]

(ii) Write down Gr_0 and run the procedure on it. It returns `OPEN`: give the setting, and say which clauses it satisfies and which it does not. Then say which ground instances of Gr_1 close the search, and by which forced moves. [1]

A4. The procedure, and a question it cannot answer.

The search of A3, run level by level, is a procedure. On a clause set $\mathcal{S}$ with variables it is

```
procedure REFUTE(S)
    k := 0
    loop:
        if DPLL(GROUND(S, k)) is CLOSED(T): return UNSAT   -- the instances at level k
        k := k + 1
```

and it has no other exit. By Herbrand's theorem it returns `UNSAT` whenever $\mathcal{S}$ is unsatisfiable, that is, whenever the claim follows; when $\mathcal{S}$ has a model, no level ever closes, and `REFUTE` runs for ever. A *semi-decision procedure* for a property says "yes" on every input that has the property and never says it on one that lacks it; `REFUTE` is one for "this does not have a model". From here on you are on your own: each part says only what to find.

(a) Trust passes along the network: $\mathsf{ok}(x)$ says that node x is trusted, and $\mathsf{nx}(x)$ is the node after x. Alice is trusted, a trusted node passes trust to the next, and Mallory claims that the third node after Alice is not trusted. The clauses are

$$\mathsf{ok}(\mathsf{alice}), \qquad \neg\mathsf{ok}(x) \vee \mathsf{ok}(\mathsf{nx}(x)), \qquad \neg\mathsf{ok}(\mathsf{nx}(\mathsf{nx}(\mathsf{nx}(\mathsf{alice})))).$$

(i) Run `REFUTE`: at each level, write the instances of the second clause and the forced moves, and say what the procedure returns there. At which level does it close? [1]

(ii) Say why no earlier level could close, and what the setting returned at level 1 says about Mallory's claim. [1]

(b) Mallory sends Charlie a rule and a question. The rule: "every key has a master key, which opens it", $\forall x\, \exists y\, \mathsf{opens}(y, x)$. The question: "does it follow that there is a key that opens every key?", $\exists y\, \forall x\, \mathsf{opens}(y, x)$. The clause form of the rule and the negated question is

$$\mathsf{opens}(m(x), x) \qquad \text{and} \qquad \neg\mathsf{opens}(y, g(y)),$$

with Skolem functions m and g.

(i) Say what $m(x)$ and $g(y)$ stand for. The clause set has no constant, so one, k, is added: write down H_0 and H_1, and count H_2. [1]

(ii) Show, by looking at the shapes of their ground instances, that no ground instance of the first clause ever clashes with one of the second, at any level. What does `REFUTE` do? So what is the answer to Mallory's question, and what is the evidence for it? Give a world with as few keys as you can. [1]

(c) Charlie has watched `REFUTE` run on Mallory's question for an hour. Why can he not conclude anything from the silence, however long it lasts? Suggest a second search that would have answered within seconds, and say what question neither search can settle. [1]

Hint: a world with exactly n keys is a finite table, and a question about a finite table is a question about switches.

(d) The clause set of A3(b) has no function symbol. Explain why, on such a clause set, a single run of `DPLL` settles the question either way, although `REFUTE` as written would still run for ever on one that has a model. [1]

Charlie's search is real. The search of A2 is the core of every SAT solver in use. MiniSat (Eén and Sörensson, 2003) fits in a few hundred lines and settles instances with hundreds of thousands of switches, because it adds one thing to the procedure: at each dead branch it *learns* a clause and keeps it. Chip designers use such solvers to check that two circuits compute the same function, and *bounded model checking* (Biere, Cimatti, Clarke and Zhu, 1999) checks a program or a chip against a requirement by unrolling it for a fixed number of steps and asking a solver whether a bad state is reachable. SMT (satisfiability modulo theories) solvers such as Z3 (de Moura and Bjørner, Microsoft Research, 2008) run the same search, treating each arithmetic fact as a switch and consulting a decision procedure for arithmetic at every step; Z3 is the engine behind Amazon's Zelkova, which checks real access policies by asking questions like the auditor's in A3(b), and behind the program verifier Dafny. First-order provers such as Vampire and E still run the search of A3 and A4, with unification choosing the instances, and are given the problems that solvers for switches cannot read. They share the limit of the procedure of A4: when such a prover has run for an hour on a claim, the user does not know whether the claim is hard or false, and so runs a model finder alongside it.

Part B — Practice

Practice, for when you want more of the same; nothing here is marked. The drills go from reading clauses to running the search. The last two, B11 and B12, go beyond this week: try them only if you are curious. In the drills, p, q, r, s, t are letters, P, Q, R are relation symbols, c and d are names, f, g, m are function symbols, and x, y, z are variables. The definitions you need:

Clauses. A literal is a letter or a negated letter; a clause is a disjunction of literals, read as "at least one" (it is also treated as the set of its literals); $\Box$ is the empty clause. A clause with literals on n different letters forbids exactly one pattern of those switches. The translation nnf removes arrows and pushes negations down ($\mathrm{nnf}(\psi \to \chi) = \mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\chi)$, $\mathrm{nnf}(\neg(\psi \wedge \chi)) = \mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\neg\chi)$, and so on, as in A1); $\mathrm{cl}(\varphi)$ is the clause set of $\mathrm{nnf}(\varphi)$ with $\vee$ distributed over $\wedge$.

Partial valuations. Under u a clause is satisfied (a literal true), falsified (all literals false) or a unit (one literal undecided, the rest false). Unit propagation adds the undecided literal of a unit, and repeats.

DPLL. Propagate; if a clause is falsified, the branch is closed; if every clause is satisfied, the branch is finished and its literals are a setting; otherwise decide an undecided letter, on side first, and search below each side. `CLOSED` exactly when the set is unsatisfiable.

Sentences to clauses. The clause form of a set of sentences, given on this sheet, names each "someone" by a Skolem constant or by a Skolem function of the variables it depends on, and has a model exactly when the sentences do. H_k: closed terms of level at most k from the constants (one added if none) and function symbols. Gr_k: every clause at every choice of terms from H_k for its variables. Herbrand's theorem: unsatisfiable exactly when some Gr_k is. `REFUTE` runs `DPLL` on $\mathrm{Gr}_0, \mathrm{Gr}_1, \ldots$ and stops only at `CLOSED`.

B1. Reading clauses. For (i)–(iii), write the pattern the clause forbids; for (iv)–(vi), write the constraint as a clause or clauses.

(i) $\neg p \vee q \vee \neg r$
(ii) r
(iii) $\neg p \vee \neg q$
(iv) never p off and q off
(v) never p on, q on and r off
(vi) p and q always agree

(vii) What does $p \vee \neg p$ forbid, and what does $\Box$ forbid?

B2. Status. Under $u = \{p, \neg q\}$, say for each clause whether it is satisfied, falsified, a unit (and which literal it forces), or none of these.

(i) $q \vee r$
(ii) $\neg p \vee q$
(iii) $p \vee s$
(iv) $\neg p \vee r \vee s$
(v) $q \vee \neg p \vee \neg r$
(vi) $\neg q$

B3. Negation normal form. Compute $\mathrm{nnf}(\varphi)$ for each, naming the equation used at each step, and then the clause form.

(i) $\neg(p \to q)$
(ii) $p \to (q \to r)$
(iii) $\neg(p \vee \neg q)$
(iv) $(p \wedge q) \to \neg r$

B4. Clause form. Compute cl of each (they are in negation normal form), and count the clauses.

(i) $(p \wedge q) \vee r$
(ii) $(p \wedge q \wedge r) \vee \neg s$
(iii) $p \vee (q \wedge (r \vee s))$

(iv) How many clauses does $(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee \cdots \vee (p_5 \wedge q_5)$ have, and why? Tseitin's translation (a remark in Chapter 7) gives each compound subformula a new letter and three clauses saying what the letter means, plus one unit clause: how many clauses is that here? What is given up in exchange?

B5. Unit propagation. Run it on the clause set

$$C_1 = \neg p, \qquad C_2 = p \vee q, \qquad C_3 = \neg q \vee r \vee s,$$
$$C_4 = \neg r, \qquad C_5 = \neg s \vee t, \qquad C_6 = \neg t \vee \neg q,$$

taking the first unit in the numbering each time, and writing each forced literal with its clause. What does the procedure return, and was a decision needed?

B6. A setting. Run the procedure on

$$C_1 = p \vee q, \qquad C_2 = \neg p \vee r, \qquad C_3 = \neg r \vee \neg q,$$
$$C_4 = \neg p \vee q \vee s, \qquad C_5 = \neg s \vee \neg r,$$

deciding p first, on side first, and draw the tree. Read off the setting it returns, say which letters it leaves free, and check it against the five clauses.

B7. A certificate. Run the procedure on

$$C_1 = p \vee q, \quad C_2 = \neg p \vee q, \quad C_3 = \neg q \vee r, \quad C_4 = \neg q \vee \neg r,$$

deciding p first, and draw the tree. Then check the tree as a certificate: for each leaf, say which clause it falsifies and why, and for each other node, which step added it.

B8. Herbrand universe. Let $\mathcal{S}$ be the clauses $P(c)$; $\neg P(x) \vee Q(f(x), d)$; $\neg Q(y, z) \vee P(g(y))$.

(i) Write down H_0 and H_1, and count H_2 without listing it.
(ii) What is the level of $g(f(c))$? Of $f(g(d))$?
(iii) How many ground instances does each clause have at level 0? Write them down.

B9. Does it follow? Let $\mathcal{S}$ be the clauses $\neg p \vee q$; $\neg q \vee r$; $\neg r \vee \neg p \vee s$. Decide, by machine and with a justification, whether $\mathcal{S}$ entails each of the clauses

(i) $\neg p \vee s$ (ii) $\neg p \vee r$ (iii) $p \vee s$,

that is, whether every setting that obeys $\mathcal{S}$ obeys the clause.

B10. Swapping quantifiers, by machine. Decide, with a justification, whether each entailment holds. The clause form of the premiss and the negated claim is given beside each.

(i) $\exists x\, \forall y\, R(x,y) \models \forall y\, \exists x\, R(x,y)$: the clauses $R(c,y)$ and $\neg R(x,d)$.
(ii) $\forall y\, \exists x\, R(x,y) \models \exists x\, \forall y\, R(x,y)$: the clauses $R(f(y),y)$ and $\neg R(x,g(x))$.

B11. *(For the curious.)* A closed tree is a proof. The *resolvent* of two clauses $p \vee C$ and $\neg p \vee D$ on p is $C \vee D$; a *resolution refutation* of $\mathcal{S}$ is a tree of clauses, each leaf a clause of $\mathcal{S}$ and each other node the resolvent of its two children, ending in $\square$. Label the closed tree of B7: each leaf with the clause it falsifies; the node above a forced node with the resolvent of the forced node's label and the clause that forced it, on the forced letter; and the node above a decision with the resolvent of the labels of its two children, on the decided letter. Draw the refutation that results and count its steps.

B12. *(For the curious.)* Keys again, with three constraints: every key has a master key, $\forall x\, \exists y\, \mathsf{opens}(y, x)$; no key opens itself, $\forall x\, \neg \mathsf{opens}(x, x)$; and if x opens y and y opens z then x opens z.

(i) Give a world that satisfies all three constraints.
(ii) Show that no world with finitely many keys obeys them.
(iii) What do `REFUTE` and the search for small structures of A4(c) each do on these three constraints, and why is each right to?