

Solving problems with SMT

A practical guide to modelling and solving with SAT + EUF + LRA + Nelson-Oppen

Copyright © 2026 by Streck.ai

Preface

If you have read the companion book that builds the solver --- *Satisfiability Modulo Theories, Phases 1-4* --- you have a solver in your hands and a fairly good idea of what is happening inside it when you call `smt_check`. This book is about how to *use* it: how to translate a problem you have into a formula the solver can answer, how to read what it tells you back, and where the edges of what it can do actually are.

The arc of the book mirrors the arc of *Predictive Algorithms*: we begin with the smallest possible working examples, build a mental model of what the solver is doing as it searches, and finish with a real worked problem --- this time, the placement of widgets in a user interface. UI placement turns out to be a good showpiece for SMT because it sits exactly in the seam between two kinds of constraint: alignment and spacing are linear arithmetic (LRA's home turf), while non-overlap is *disjunctive* (the SAT layer's home turf). Tools that handle only one of the two --- linear solvers like Cassowary, or pure constraint-satisfaction backtrackers --- end up either over-relaxing the problem or over-discretising it. SMT handles both at once.

A short reading guide. If you already have a problem in mind, you can skip the first two chapters and jump to the modelling-patterns chapter (4). If you want to understand what the solver is doing inside, the companion book is the right place; chapter 3 here gives only enough of the mental model to make the patterns and worked examples land.

We assume you have built the solver via `./build.zsh` and seen its sanity tests pass; that puts a `build/` directory next to your sources with a `smt.o` object file you can link against. All code in this book is written against the public interface in `smt.h`; nothing here pokes at solver internals.

— Stockholm, May 2026

1. Quick start

The smallest interesting interaction with the solver is three lines:

```
SmtCtx *ctx = smt_new();
smt_assert_eq(ctx, smt_mk_const(ctx, "a"), smt_mk_const(ctx, "b"));
int r = smt_check(ctx);
// r == SMT_SAT: there is a model where a and b are the same value.
smt_free(ctx);
```

Three different theories, three "hello world" problems.

1.1. Hello, EUF

Two constants a and b ; the equality $a = b$ is satisfiable.

```
#include "smt.h"
#include <stdio.h>

int main(void) {
    SmtCtx *ctx = smt_new();
    int a = smt_mk_const(ctx, "a");
    int b = smt_mk_const(ctx, "b");
    smt_assert_eq(ctx, a, b);
    printf("%d\n", smt_check(ctx));    // 10 (SMT_SAT)
    smt_free(ctx);
}
```

A slightly less trivial EUF problem: transitivity. $a = b$ and $b = c$ imply $a = c$, so asserting $a \neq c$ alongside is unsatisfiable.

```
int a = smt_mk_const(ctx, "a");
int b = smt_mk_const(ctx, "b");
int c = smt_mk_const(ctx, "c");
smt_assert_eq(ctx, a, b);
smt_assert_eq(ctx, b, c);
smt_assert_neq(ctx, a, c);
// smt_check returns SMT_UNSAT.
```

The solver finds this contradiction without exploring any choices: the EUF theory propagates $a = c$ from the chain, the negated equality already on the trail conflicts with it, and the search ends on the first call to `euf_check`. We will come back to what "propagates" and "conflicts" mean in chapter 3.

1.2. Hello, LRA

Real-valued variables and linear inequalities. $x + y \leq 5$ with $x > 3$ and $y > 3$ is unsatisfiable, because the two lower bounds together give $x + y > 6$, which contradicts the upper bound on the sum.

```
int x = smt_mk_real_var(ctx, "x");
int y = smt_mk_real_var(ctx, "y");

SmtLinTerm xy[2] = { {x, 1, 1}, {y, 1, 1} };    // 1*x + 1*y
smt_assert_lra_le(ctx, xy, 2, 5, 1);           // x + y <= 5
```

```

// x > 3 is the negation of x <= 3.
SmtLinTerm xt = { x, 1, 1 };
int a_x_le_3 = smt_mk_lra_le_atom(ctx, &xt, 1, 3, 1);
int unit = -a_x_le_3;
smt_assert_clause(ctx, &unit, 1);           // x > 3

// Same for y.
SmtLinTerm yt = { y, 1, 1 };
int a_y_le_3 = smt_mk_lra_le_atom(ctx, &yt, 1, 3, 1);
int unit = -a_y_le_3;
smt_assert_clause(ctx, &unit, 1);           // y > 3

// smt_check returns SMT_UNSAT.

```

Two things to notice in the API. First, the LRA assertion functions take explicit coefficients and a right-hand-side as separate numerator/denominator pairs --- the solver is exact-rational throughout; there is no floating-point representation of any constant. Second, there is no "assert greater than"; you make a strict-less-than atom and assert its negation. The solver's atoms only ever face one way (\leq or $<$); the opposite direction is the negated literal in a unit clause. The reasons for this design are explained in chapter 3, but the practical takeaway is that you assemble atoms and then assemble clauses out of them.

1.3. Hello, Nelson-Oppen

`SmtShared` is the API for variables that participate in *both* theories. The classic Nelson-Oppen example: $f(x) \neq f(y)$ and $x = y$, where x and y are shared variables, is unsatisfiable. EUF receives the equality, merges the classes, congruence on f gives $f(x) = f(y)$, and that contradicts the inequality.

```

SmtShared x = smt_mk_shared(ctx, "x");
SmtShared y = smt_mk_shared(ctx, "y");
int fx = smt_mk_app(ctx, "f", &x.euf, 1);
int fy = smt_mk_app(ctx, "f", &y.euf, 1);
smt_assert_neq(ctx, fx, fy);
smt_assert_shared_eq(ctx, x, y);
// smt_check returns SMT_UNSAT.

```

The interesting case is the *other* direction: $x = 5, y = 5, f(x) \neq f(y)$. LRA pins both x and y to the same value, propagates the bridge atoms back to EUF (more on that in chapters 3 and 4), and the contradiction closes again. From the API side it just looks like:

```

SmtLinTerm xt = { x.lra, 1, 1 };
SmtLinTerm yt = { y.lra, 1, 1 };
smt_assert_lra_eq(ctx, &xt, 1, 5, 1);
smt_assert_lra_eq(ctx, &yt, 1, 5, 1);
smt_assert_neq(ctx, fx, fy);
// UNSAT.

```

2. The shape of the API

There are five kinds of object the user constructs.

SmtCtx is the solver state. One per problem instance; you create it, build a formula on it, call `smt_check`, optionally inspect the model, and free it. The solver is one-shot in Phases 1-4: there is no push/pop API and calling `smt_check` twice on the same context is not supported. You make a new context per problem.

Terms (type `SmtTerm`, an opaque integer id) belong to EUF. They are constants made with `smt_mk_const`, or function applications made with `smt_mk_app`. Application terms are hash-consed: `smt_mk_app(f, [a, b])` always returns the same `SmtTerm` no matter how many times you call it. This matters for congruence reasoning --- the solver discovers that $f(a)$ and $f(b)$ are the same term as soon as it knows $a = b$, and the hash-consing is what makes that discovery a single union-find merge instead of a search.

Variables (type `SmtVar`, also an opaque integer id) belong to LRA. They are real-valued and made with `smt_mk_real_var`. The two namespaces do not overlap; an `SmtTerm` is not interchangeable with an `SmtVar` even if their underlying integer ids happen to match.

Shareds (type `SmtShared`, a struct of `{ euf, lra }`) live in both worlds at once. They are how mixed-sort terms get into the solver: a shared variable can be the argument of an EUF function and at the same time take part in an LRA constraint. The two views are kept consistent by the Nelson-Oppen bridges installed at `smt_mk_shared` time.

Atoms (type `SmtAtom`, again an integer id) are Boolean propositions the SAT layer can hold a truth value for. There are three constructors:

- `smt_mk_eq_atom(a, b)` builds the EUF atom $a == b$. - `smt_mk_lra_le_atom(terms, n, rhs_num, rhs_den)` builds $\text{sum}(c*x) \leq r$. - `smt_mk_lra_lt_atom(...)` builds $\text{sum}(c*x) < r$.

Atoms are 1-indexed positive integers. A *literal* is a signed atom: `+atom` for the atom asserted true, `-atom` for it asserted false. Clauses are arrays of literals plus a length. The shape of a unit assertion is therefore just "make an atom, put its id (with sign) in a 1-element array, push the clause."

There are convenience shortcuts that bundle atom creation with a unit-clause assertion: `smt_assert_eq`, `smt_assert_neq`, `smt_assert_lra_le`, `smt_assert_lra_lt`, `smt_assert_lra_eq`, `smt_assert_shared_eq`, `smt_assert_shared_neq`. These are what you write when you do not need the atom for anything else (e.g. you are not also putting it in a larger clause). For any *disjunctive* assertion you build the atoms first and then call `smt_assert_clause` with the resulting literal array.

3. A mental model of what the solver is doing

You do not need to know how the solver works in order to use it on small problems, but the moment something is unexpectedly slow, returns SAT when you expected UNSAT (or vice versa), or just behaves in a way you cannot explain, the mental model pays off. This chapter sketches the model. The companion book has the full version with code; this is the user-facing version: enough to read the statistics and reason about modelling choices, no more.

3.1. The two-level structure

There are two solvers inside, stacked.

At the bottom, a **SAT solver** sees only literals. It has no idea what the atoms mean. It does propositional reasoning --- unit propagation, clause learning, backjumping --- on a flat universe of Boolean propositions.

Above that, **theory solvers** (EUF and LRA in our case) understand what the atoms actually mean and check consistency on whatever subset of atoms is currently true on the SAT trail. When the SAT solver picks a partial assignment, the theory looks at it and reports back: *consistent* (search deeper or declare SAT if the trail is complete), *inconsistent* (here is a clause naming a subset of asserted literals that cannot all be true --- the SAT solver learns this clause as a new constraint), or *I have implications* (here is a literal the theory has derived; please assign it on the trail before deciding further).

That third channel --- theory propagation --- is what powers Nelson-Oppen. Each theory advertises implied literals, the SAT solver pushes them onto the trail, BCP carries them through ordinary clauses, and the next theory sees the consequences. The implementation book describes the channel and its reasons in detail; for modelling purposes, the consequence is simple: **you do not have to manually thread information between theories**. Setting up the right atoms and clauses is enough; the propagation channel does the threading.

3.2. What the statistics mean

After `smt_check` you can ask the context for several counters:

- `smt_decisions` --- how many literals the SAT solver had to guess. - `smt_conflicts` --- how many learned-clause cycles ran. - `smt_theory_conflicts` --- how many of those conflicts came from a theory rather than from BCP itself. - `smt_theory_props` --- how many literals a theory pushed onto the trail in advance of a SAT decision. - `smt_lra_pivots` --- how many simplex pivots the LRA solver did.

These numbers tell you which layer is doing the work. A pure-LRA problem with `decisions = 0` is one the simplex tableau solves directly without any propositional case analysis. An EUF problem with `theory_props > 0` is one where the congruence-closure structure forced equalities the user did not assert. A Nelson-Oppen problem where you see `theory_props` from LRA and a `theory_conflict` from EUF is the full back-and-forth working as intended.

If you see a huge number of decisions for a problem you expected to be tractable, that is a signal the disjunctive structure of the formula is larger than the theory side. Reducing disjunctions --- often by making implicit ordering constraints explicit, as in section 4.5 below --- is usually the right response.

3.3. The two LRA atom kinds and what they imply

The LRA front-end has exactly two atom shapes: \leq and $<$. Equality $a = b$ is asserted as the *conjunction* of $a \leq b$ and $b \leq a$, internally via two atoms; the convenience function `smt_assert_lra_eq` does this for you. The four polarities of the two atom kinds cover every inequality you might want:

- + atom of \leq says $\text{expr} \leq r$ (weak upper bound) - - atom of \leq says $\text{expr} > r$ (strict lower bound)
- + atom of $<$ says $\text{expr} < r$ (strict upper bound) - - atom of $<$ says $\text{expr} \geq r$ (weak lower bound)

If you want to say $\text{expr} \geq r$, build the $<$ atom for $\text{expr} < r$ and assert its negation. This feels backwards at first; with a few problems it becomes second nature, and the symmetry of the encoding pays off in the solver's internals (delta-rationals, see the companion book).

3.4. The four sources of "unexpected"

Roughly four kinds of surprise come up in modelling. Each has a typical cause.

Unexpected SAT. The most common reason is missing a constraint you thought you had. Re-read the formula: every atom you create is *optional* unless you put it in a unit clause or assert it via one of the `smt_assert_*` shortcuts. An atom that exists but is not asserted is just a proposition the SAT solver will freely set true or false.

Unexpected UNSAT. Usually means two constraints are tighter than you realised. A common culprit in LRA modelling is a strict inequality where you meant a weak one (or vice versa). The four-polarity table above is the place to start.

Unexpected slowness. Almost always one of two things. Either there are many disjunctive constraints (non-overlap of N boxes is $N*(N-1)/2$ four-literal clauses) and the search space is genuinely big; or you have inadvertently asked LRA to handle a constraint that needs SAT case analysis (e.g. a non-convex region encoded as a single inequality). Statistics tell you which: many decisions = SAT-side, many pivots without many decisions = LRA-side.

Unexpected `unknown`. The solver returns this only when you have already called `smt_check` once and called it again on the same context. Make a fresh `SmtCtx`.

4. Modelling patterns

Each section here shows a small pattern with the C code, the formula it expresses, and the typical statistics profile when the solver runs it. The patterns build on each other; the UI worked examples in chapter 5 use them in combination.

4.1. The unit assertion

The simplest atom-and-clause combination is asserting an atom alone:

```
int a = smt_mk_const(ctx, "a");
int b = smt_mk_const(ctx, "b");
smt_assert_eq(ctx, a, b);           // shortcut
```

or equivalently:

```
int atom = smt_mk_eq_atom(ctx, a, b);
int unit = atom;                    // +atom means "atom is true"
smt_assert_clause(ctx, &unit, 1);
```

The shortcuts (`smt_assert_eq`, `smt_assert_lra_le`, ...) exist precisely to skip the second form for the common case. Use them by default; reach for the explicit form only when you also want the atom id to put in another clause.

4.2. Negation by sign flip

The two ways to make $a \neq b$ look like this:

```
smt_assert_neq(ctx, a, b);          // shortcut

int atom = smt_mk_eq_atom(ctx, a, b);
int unit = -atom;
smt_assert_clause(ctx, &unit, 1);    // -atom means "atom is false"
```

This is what is happening when you assert $x > 3$: there is no `smt_assert_lra_gt`, because $x > 3$ is just $-(x \leq 3)$:

```
SmtLinTerm xt = { x, 1, 1 };
int a_x_le_3 = smt_mk_lra_le_atom(ctx, &xt, 1, 3, 1);
int unit = -a_x_le_3;
smt_assert_clause(ctx, &unit, 1);
```

4.3. Equality as two inequalities

$\text{expr} == r$ is $\text{expr} \leq r$ and $\text{expr} \geq r$. The first is one atom; the second is the negation of $\text{expr} < r$. The library provides `smt_assert_lra_eq` that does both:

```
SmtLinTerm xt = { x, 1, 1 };
smt_assert_lra_eq(ctx, &xt, 1, 5, 1); // x == 5
```


Internally that produces two atoms ($x \leq 5$ and $x < 5$) and two unit clauses ($+a_le$ and $-a_lt$). If you want the equality to be one atom of a larger clause (a disjunction like " $x == 5$ or $y == 7$ "), you have to build the two-atom-and-clause structure by hand. The example in section 5.4 does this.

4.4. Case analysis as a clause

A disjunction is just a clause with more than one literal. The non-overlap constraint between two horizontal rectangles is the canonical example:

```
// A is to the left of B    OR    B is to the left of A.
SmtLinTerm a_minus_b[2] = { { ax, 1, 1 }, { bx, -1, 1 } };
SmtLinTerm b_minus_a[2] = { { bx, 1, 1 }, { ax, -1, 1 } };
int a_left = smt_mk_lra_le_atom(ctx, a_minus_b, 2, -BOX_W, 1);
int b_left = smt_mk_lra_le_atom(ctx, b_minus_a, 2, -BOX_W, 1);
int cl[2] = { a_left, b_left };
smt_assert_clause(ctx, cl, 2);
```

Each branch of the disjunction is one literal in the clause. The SAT solver decides which is true; LRA either confirms or refutes the choice.

4.5. Chains beat pairwise disjunctions

If you know a *total order* on the items being placed --- "B0 is the leftmost, B1 next, then B2..." --- you can replace the pairwise non-overlap clauses with a chain of inequalities:

```
for (int i = 0; i + 1 < N; i++) {
    SmtLinTerm gap[2] = { { x[i], 1, 1 }, { x[i+1], -1, 1 } };
    smt_assert_lra_le(ctx, gap, 2, -(BOX_W + GAP), 1);    // x[i] + BOX_W +
GAP <= x[i+1]
}
```

This is dramatically tighter than the disjunctive encoding: there are no choices for the SAT layer to make, just a sequence of bounds for LRA to enforce. Example 11 (`ex11_ui_row_layout.c`) solves four buttons in a row in 0 decisions and 4 pivots; the equivalent disjunctive formulation has six four-literal clauses and forces the SAT solver to enumerate orderings.

The reverse rule is the same: when you do *not* have a natural total order (a free 2D layout, see `ex12`) you have to bite the disjunctive bullet. Try to discover orderings in your problem and exploit them.

4.6. Linear equality and alignment

Centering, equal spacing, axis sharing --- all of these are linear equalities and live entirely inside LRA. There is no theory dispatch at all on a problem that consists only of equalities and inequalities.

Vertical centering of a button in a toolbar, for instance:

```
// y == (TOOLBAR_H - BUTTON_H) / 2
SmtLinTerm yt = { y, 1, 1 };
smt_assert_lra_eq(ctx, &yt, 1, (TOOLBAR_H - BUTTON_H) / 2, 1);
```

Equal horizontal spacing between three buttons:

```
// 2 * B1.x == B0.x + B2.x
SmtLinTerm spacing[3] = { { x1, 2, 1 }, { x0, -1, 1 }, { x2, -1, 1 } };
```

```
smt_assert_lra_eq(ctx, spacing, 3, 0, 1);
```

The coefficient field is a (num, den) pair, so fractional coefficients ("two-thirds of the parent width") are not a problem.

4.7. Shared variables across theories

Whenever a variable participates in *both* EUF (as a function argument) and LRA (as a constraint variable), make it a `SmtShared` rather than two separate objects:

```
SmtShared x = smt_mk_shared(ctx, "x");

int fx = smt_mk_app(ctx, "f", &x.euf, 1);
SmtLinTerm xt = { x.lra, 1, 1 };
smt_assert_lra_le(ctx, &xt, 1, 10, 1);
```

The `smt_mk_shared` call installs cross-theory "bridge" atoms (eq, le, ge) and linking clauses for every pair of existing shares. Those bridges are what carry derived equalities between the two theories. They do nothing unless they fire --- they are purely additional propagation channels --- so the cost is real but bounded. (See chapter 6 for the asymptotic.)

4.8. Reusing atoms across clauses

If the same atom appears in multiple clauses, build it once and reuse the id. The atom-construction functions deduplicate equality atoms (the same (a, b) pair returns the same `SmtAtom`), but they do *not* deduplicate LRA atoms. Constructing the same $x + y \leq 5$ twice gives two atoms; the solver treats them as independent propositions and will not propagate one from the other.

In practice this means: if your problem has natural recurring combinations (think " $B[i].x - B[j].x \leq -BOX_W$ " in a non-overlap encoding), store the atom ids and reuse them.

4.9. Soft constraints and preferences

The solver does not have a built-in notion of preference or cost. If two models satisfy the formula, the solver returns whichever it finds first. For "we would prefer this layout but any valid one is acceptable", you have two options:

- **Tighten the bounds.** Replace " $x \geq 0$ " with " $x \geq 5$ " if 5 is your preferred minimum margin; if that turns out to be infeasible the solver will tell you (UNSAT) and you can relax.

- **Solve incrementally.** With no push/pop API in Phases 1-4, this means making a fresh `SmtCtx` per attempt. It is more verbose but works fine for the kinds of problems this solver is sized for.

A general optimisation layer (MaxSMT, optimisation modulo theories) is not part of the Phase 1-4 implementation.

4.10. Common pitfalls

A short list of mistakes you will probably make at least once.

Forgetting to assert an atom you constructed. Atoms created by `smt_mk_*` are not asserted; they are just registered. They only become constraints when you put them in a clause. The SAT solver may freely choose them either way.

Strict vs weak in the wrong direction. When you negate a $<$ atom you get \geq , not $>$. When you negate a \leq atom you get $>$, not \geq . The table in section 3.3 is the canonical reference; print it out.

Mixing namespaces. An `SmtVar` (LRA) cannot be the argument of an EUF function. If you need that, make the variable a `SmtShared` from the start. Trying to retrofit a shared variable in the middle of a problem build is not supported in Phases 1-4.

Integer overflow on coefficients. The internal rational type is int64 numerator over int64 denominator. For tutorial-sized problems this is fine; for adversarial inputs --- a chain of pivots that doubles the numerator each step --- it can overflow. If you are pushing the solver's size, normalise your input coefficients aggressively (divide by gcd, pick the smallest representative) before feeding them in.

5. Worked example: UI component placement

Here is the test problem: given a canvas of width W and height H , plus a set of UI elements with fixed sizes and a list of layout constraints, find a position for every element such that all constraints are satisfied.

This problem comes up in three places in practice:

1. **Auto-layout engines** in Cocoa and Android. The Cassowary algorithm --- an incremental dual-simplex specialised for soft and required linear constraints --- is the widely-cited reference, and while Apple's Auto Layout is closed-source the two share concepts. The constraint vocabulary is rich (equalities with priorities, multipliers, "greater than or equal" with a target) but fundamentally linear --- no disjunctive non-overlap, no case analysis.

2. **Graph drawing.** Force-directed layout is the popular technique, but for problems where exact placement matters (chip floorplanning, PCB layout, document layout with hard constraints) constraint solvers are used. The classic graphviz `dot` driver uses simplex internally.

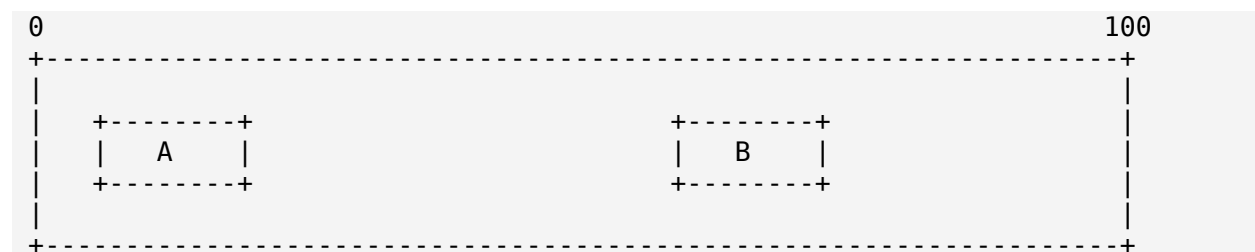
3. **CSS layout.** Flexbox and grid are themselves constraint solvers, hand-coded for speed. They handle the common cases very well and drop to inline expressions for the edge cases.

SMT sits one level above the linear approaches. It can do everything they can (linear inequalities and equalities, alignment, spacing) and also handle the disjunctive constraints they cannot (non-overlap of rectangles, "at least one of these edges aligns", "exactly one of these rows is used"). The cost is a more verbose API and a runtime cost that grows with the disjunctive depth of the problem.

This chapter walks four worked examples from the `examples/` directory. Each is a complete, runnable C program; this text annotates the design choices and reads the statistics.

5.1. Two boxes in a row (`ex10_ui_two_boxes.c`)

The smallest non-trivial layout problem: two rectangles A and B, each 40 wide, in a canvas 100 wide, must not overlap.



The formula has two free variables (the x-coordinates of A and B) and five constraints: A inside the canvas (two bounds), B inside the canvas (two bounds), and the non-overlap disjunction.

The non-overlap is the only piece worth dwelling on. Two horizontal rectangles fail to overlap iff one is entirely left of the other:

$$A.\text{right} \leq B.\text{left} \quad \text{OR} \quad B.\text{right} \leq A.\text{left}$$

Substituting $\text{right} = \text{left} + \text{width}$ and rearranging:

$$A.x - B.x \leq -40 \quad \text{OR} \quad B.x - A.x \leq -40$$

That gives a clause with two LRA atoms:

```

SmtLinTerm a_minus_b[2] = { { ax, 1, 1 }, { bx, -1, 1 } };
SmtLinTerm b_minus_a[2] = { { bx, 1, 1 }, { ax, -1, 1 } };
int a_left = smt_mk_lra_le_atom(ctx, a_minus_b, 2, -BOX_W, 1);
int b_left = smt_mk_lra_le_atom(ctx, b_minus_a, 2, -BOX_W, 1);
int cl[2] = { a_left, b_left };
smt_assert_clause(ctx, cl, 2);

```

The solver's output:

```

canvas width    = 100
box width       = 40
result          = SAT (valid layout exists)
decisions       = 1
theory conf.    = 0
LRA pivots      = 2

```

One SAT decision (the disjunction had to be resolved one way or the other), no theory conflicts (the chosen branch was feasible), two LRA pivots to find the actual values. The two coefficients $+1$ and -1 on each atom are what put $B.x$ and $A.x$ in the tableau; the pivots position them.

5.2. Four buttons in a row (`ex11_ui_row_layout.c`)

If the layout is naturally ordered --- "B0 then B1 then B2 then B3" --- the disjunctive non-overlap can be replaced with a chain of inequalities. This is the single most important optimisation in SMT-based UI layout. For a row of N buttons, the disjunctive form has $N(N-1)/2$ clauses of four literals each (for 2D) or two literals each (for 1D); the chain form has $N-1$ clauses with one literal each.

```

for (int i = 0; i + 1 < N; i++) {
    SmtLinTerm gap[2] = { { x[i], 1, 1 }, { x[i+1], -1, 1 } };
    smt_assert_lra_le(ctx, gap, 2, -(BOX_W + GAP), 1);
}

```

The constraint says $x[i] + BOX_W + GAP \leq x[i+1]$ for each adjacent pair: $B[i]$ ends, a gap of GAP follows, then $B[i+1]$ starts.

Plus the row-fits-canvas constraint:

```

SmtLinTerm last = { x[N-1], 1, 1 };
smt_assert_lra_le(ctx, &last, 1, CANVAS_W - BOX_W, 1);

```

And the row-starts-at-or-after-zero:

```

SmtLinTerm neg0 = { x[0], -1, 1 };
smt_assert_lra_le(ctx, &neg0, 1, 0, 1);

```

Output:

```

buttons        = 4
result          = SAT (valid layout)
decisions       = 0 (expect 0: pure chain, no disjunction)
theory conf.    = 0
LRA pivots      = 4

```

Zero decisions. The SAT layer has nothing to do: every clause is a unit clause and BCP places every literal on the trail before any decision is needed. LRA does four pivots to actually position the buttons.

The price of the chain form is that you have committed to an order. If the row has to lay out elements whose left-to-right order is unknown --- say, dragging a column reorders them --- the chain form does not fit and you need the disjunctive encoding.

5.3. Four free 2D boxes (`ex12_ui_grid_placement.c`)

The general case. Four 40x30 boxes in a 100x100 canvas, no fixed ordering, no overlap. Each pair contributes a four-literal clause: A is left of B, or B is left of A, or A is above B, or B is above A.

```
for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
        /* ... build left, right, above, below atoms ... */
        int cl[4] = { left, right, above, below };
        smt_assert_clause(ctx, cl, 4);
    }
}
```

Six pairs, six four-literal clauses, twenty-four atoms plus the box-in- canvas bounds. The solver:

| | |
|--------------|--|
| canvas | = 100 x 100 |
| box | = 40 x 30 |
| pairs | = 6 (each: 4-literal non-overlap clause) |
| result | = SAT (valid layout) |
| atoms | = 40 |
| decisions | = 31 |
| theory conf. | = 6 |
| conflicts | = 6 |

31 decisions, 6 theory conflicts. The SAT solver enumerates orderings; some of those are infeasible (the boxes do not fit in that order) and LRA learns clauses ruling them out. The solver finds a valid layout after the sixth backjump.

If you want every box positioned in a specific *region* of the canvas ("box 0 is in the top-left quadrant, box 1 in the top-right..."), add bounds to constrain each box's coordinates and the search collapses -- the inequalities are essentially picking the order for the SAT layer.

5.4. A real toolbar (`ex13_ui_toolbar.c`)

Three icon buttons (24x24) in a horizontal toolbar (240x40) with the constraints any reasonable toolbar imposes:

1. Buttons fit inside the toolbar (with 8px padding).
2. Buttons don't overlap.
3. Buttons are vertically centered ($y = 8$ for each).
4. Equal horizontal spacing: $\text{gap}(B_0, B_1) = \text{gap}(B_1, B_2)$.
5. Left-to-right ordering: B_0 left of B_1 left of B_2 .

Vertical centering (constraint 3) is the cleanest: a linear equality.

```
for (int i = 0; i < 3; i++) {
    SmtLinTerm yt = { ys[i], 1, 1 };
    smt_assert_lra_eq(ctx, &yt, 1, (TOOLBAR_H - ICON) / 2, 1);
}
```

The ordering (5) replaces the disjunctive non-overlap with a chain:

```
SmtLinTerm chain01[2] = { { x0, 1, 1 }, { x1, -1, 1 } };
SmtLinTerm chain12[2] = { { x1, 1, 1 }, { x2, -1, 1 } };
smt_assert_lra_le(ctx, chain01, 2, -ICON, 1);
smt_assert_lra_le(ctx, chain12, 2, -ICON, 1);
```

And the equal-spacing (4) is one linear equality:

```
// gap(B0,B1) == gap(B1,B2) <=> B1.x - (B0.x + ICON) == B2.x - (B1.x +
ICON)
// <=> 2 B1.x - B0.x - B2.x == 0
SmtLinTerm spacing[3] = { { x1, 2, 1 }, { x0, -1, 1 }, { x2, -1, 1 } };
smt_assert_lra_eq(ctx, spacing, 3, 0, 1);
```

Output:

```
toolbar      = 240 x 40
icon         = 24 x 24
padding      = 8
result       = SAT (valid layout)
decisions    = 0
theory conf. = 0
LRA pivots   = 5
```

Zero decisions. The toolbar has structure: alignment is a linear equality, ordering is a chain of inequalities, equal-spacing is a linear equality. None of those need SAT case analysis. The whole problem fits inside the simplex tableau and the solver dispatches it in five pivots.

That is the take-home message of this chapter. When you can express structural design intent --- alignment, ordering, equal-spacing, centering --- as linear equalities, the solver does very little propositional work. Reach for disjunctive non-overlap only when the problem genuinely is not ordered (free placement, drag-and-drop, etc.).

6. What the solver can and cannot do

A solver this small is a tool with sharp edges. Use it for what it is good for; reach for something else when the problem is outside its range.

6.1. Inside the comfortable range

- **Tens to low hundreds of variables.** All the worked examples in this book fall well below the upper bound. EUF problems with a few dozen terms and a few hundred atoms run in milliseconds. LRA problems of similar size run in a small number of pivots.

- **Convex linear arithmetic.** LRA in the Dutertre-deMoura form is a clean, fast solver for the convex linear-rational problems UI layout, network flow, and basic resource allocation produce. Mixed EUF / LRA via Nelson-Oppen handles formulas where the two interact naturally.

- **Boolean structure over theory atoms.** This is the *defining* advantage of SMT over plain LP: disjunctions of inequalities, conditional constraints ("if this Boolean variable is true then this LRA constraint applies"), case-by-case design rules. The SAT layer handles the case analysis; the theory handles each case.

- **Exact rational arithmetic.** No floating-point rounding. Two rationals you compute will compare exactly equal if they mathematically are, regardless of the path you took to compute them.

6.2. Outside the comfortable range

- **Integer arithmetic.** The solver handles linear *rational* arithmetic; integer variables would require linear integer arithmetic (LIA), which is non-convex (the equality $x = y/2$ has integer solutions only for some values of y). Nelson--Oppen for LIA needs case-splitting inside the theory and is a substantially larger build. If you need integers, model them as rationals plus a "good enough" rounding pass on the output, or step up to a production solver like cvc5 or Z3.

- **Optimisation.** The solver returns a satisfying assignment if one exists, but it does not pick the *best* among many. MaxSMT (find an assignment maximising the number of satisfied soft clauses) and OMT (optimisation modulo theories) are extensions you can read about in the literature but they are not in this implementation.

- **Very large formulas.** The int64 rational backing store, the eager Nelson--Oppen bridge atoms (quadratic in the shared-variable count), and the dense tableau representation all rule out problems in the thousands-of-variables range. The solver is a tutorial artefact and is sized accordingly.

- **Non-linear arithmetic.** Multiplying two variables is outside LRA. There are SMT solvers for non-linear real arithmetic (the NRA theory), but they are vastly more complex than LRA and not in this implementation.

- **Quantifiers.** Everything in this solver is quantifier-free. The common quantified extensions (E-matching for instantiating quantified EUF axioms, MBQI for model-based quantifier instantiation) are not here either.

6.3. Performance signposts

Some rough numbers from the examples in this book, running on a 2024-era laptop:

- Pure chains of LRA inequalities (ex11, ex13): well under a millisecond. - Disjunctive non-overlap of N rectangles (ex12): scales roughly with N^2 atoms and an exponential worst-case for the SAT search, though the actual cases that arise in UI layout are mild. - Mixed EUF + LRA via Nelson--Oppen with a handful of shared variables (ex8, ex9): also sub-millisecond.

If a problem you have is taking unreasonably long, the statistics will tell you why. Many decisions = the SAT layer is enumerating; many pivots without many decisions = the simplex is doing work; many theory propagations = the cross-theory bridges are firing. Each suggests a different remediation: reduce disjunctions, simplify the linear part, or reduce the number of shareds, respectively.

6.4. When to reach for a different tool

- **For pure linear constraints with priorities** (typical UI auto-layout): use Cassowary or any LP solver. It will be faster and the API is closer to what you want.

- **For pure non-overlap of many rectangles** (chip placement, dense UI grids): a specialised geometric algorithm (corner stitching, binary space partitioning) will beat SMT decisively at the size where SMT starts to struggle.

- **For mixed linear + Boolean + integer** at industrial scale: cvc5 or Z3. Both are production-grade and handle every theory mentioned in this book and many more, with SMT-LIB v2 as the input language.

The reason to use the solver built across Phases 1-4 is the same as the reason to read the implementation book: to understand exactly what is happening, to be able to modify the algorithm when your problem has exotic structure, and to have something small and dependency-free for embedding in another C codebase. For workloads at that scale and shape, it is the right tool.

7. Further reading

- **The companion book**, *Satisfiability Modulo Theories, Phases 1-4*. The implementation side of everything in this guide.
- Daniel Kroening and Ofer Strichman, *Decision Procedures: An Algorithmic Point of View*, 2nd ed., Springer 2016. The standard textbook treatment of EUF, DPLL(T), and combined theories.
- Bruno Dutertre and Leonardo de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," CAV 2006. The primary reference for the LRA algorithm in this solver.
- Greg Badros, Alan Borning, and Peter Stuckey, "The Cassowary Linear Arithmetic Constraint Solving Algorithm," ACM TOCHI 8(4), 2001. The linear-arithmetic solver behind Auto Layout-style UI engines. Reading this alongside Dutertre and de Moura is the fastest way to understand the design space of solvers for layout-like problems.
- The source of cvc5 (<https://cvc5.github.io>) and Z3 (<https://github.com/Z3Prover/z3>). The reference implementations.
- Clark Barrett, Pascal Fontaine, and Cesare Tinelli, *The SMT-LIB Standard, Version 2.6*, 2017. The input format you would target if you wanted to add a parser to this solver.