Constraints and Constraint Solvers in ThingLab

Alan Borning

Draft of November 12, 2026

1 Introduction

ThingLab was a system I built for my Ph.D. dissertation research. It was a constraint-oriented simulation laboratory, originally implemented in Smalltalk-76, that let the user build graphical objects representing such things as electrical circuits, geometric constructions, or documents with constraints on layout and contents. After experimenting with electrical circuits, I realized the same idea could be used for programming language elements (e.g., rather than a resistor with two leads that enforces an Ohm's Law constraint, you could have a Plus object with three leads terminating in numbers, which would enforce a plus constraint among the three numbers).

In summer 2026, I implemented a reproduction of the original system, written in TypeScript and running in a web browser. This started as a small experiment in trying out AI-assisted software development but got out of hand. The reproduction is intended to keep the look and feel of the original, but (a) to work reliably — for example, the implementation of the relaxation algorithm in the original had numerical stability problems if you made too large a change — and (b) to incorporate subsequent research on formalizing hard and soft constraints, on declaratively describing change over time, and on designing and implementing new constraint solvers that found solutions that were correct according to those formalisms. The reproduction currently includes three constraint solvers: DeltaBlue for local propagation, Cassowary for linear equalities and inequalities, and relaxation for other numeric constraints, along with a cooperating solvers framework that partitions the constraints among the solvers and handles communication among them.

The architecture for cooperating solvers is new and will hopefully be useful for modern UI applications as well. For readers who are familiar with the background and just want to read about the new cooperating solvers architecture, see Section 4. Otherwise, this document is intended to provide an overview of the overall implementation of constraints and constraint solvers in the reproduction, including the cooperating solvers architecture, and thus encompasses quite a few years of research by colleagues and students as well as myself.

Currently this is just an informal note rather than a more formal academic publication, and lacks (among other things) careful citations to related work. The reproduction itself was almost entirely written using Claude Code based on a series of detailed prompts; this note is written entirely by a human (me), except for the appendix, which I had Claude write to provide Claude's notion of how the code works.

2 Background — Constraints

Sketchpad [10, 11] pioneered constraints and solvers: constraints as relations that should be maintained, represented in a general way as error functions; relaxation as a general-purpose solver for numeric constraints;

and propagation of degrees of freedom for efficiently solving constraints that could be handled by local propagation after the central set of constraints was solved.

The original ThingLab [2, 3] added propagation of known states and methods for local propagation constraints, thus allowing non-numeric constraints, for example on strings, to be solved as well. (Actually I have heard that Sketchpad too had propagation of known states, but it's not documented in the dissertation.)

ThingLab left open a number of questions about constraint semantics, as well as highlighting the need for more efficient solvers. Consider the quadrilateral example that is shown when you start the reproduction. When you move one of the endpoints, why doesn't the figure just collapse to a single point? That would, after all, be a valid solution to the midpoint constraints. And if you add a general requirement that values should not change unnecessarily, why should the solver pick the solution it does, rather than pivoting about the midpoints? To see what the latter looks like, add anchors to each of the midpoints. Or consider the Bounded Quadrilateral example, which is the same except that the quadrilateral must lie within a bounding box. How should it behave when the user tries to drag an endpoint or a midpoint outside of the bounding box? (Try it.) Finally, it's useful to have one-way constraints, for example on the constants in the TemperatureConverter example: the user should be able to edit the text and update a constant, but solving for a new temperature shouldn't be able to change the constant.

2.1 Hard and Soft Constraints

In response to these sorts of questions, my students and other collaborators and I worked out a theory of hard and soft constraints, different ways of trading off possible solutions, and read-only annotations. Here is a brief summary; for a full description see [6]. The theory provides more satisfying answers to questions such as "why doesn't the quadrilateral collapse to a point" than "because the solver tries to leave things where they used to be." In addition, having a theory of hard and soft constraints, independent of any particular algorithm, leaves the researcher in a better position to argue (or ideally prove) that a particular constraint satisfaction algorithm gives correct results.

In the theory, there can be an arbitrary number of constraint strengths; in the reproduction, the following strengths are built in: `required`, `strong`, `medium`, `weak`, and `weakest`. Required constraints must be satisfied — it is an error if they are not — while soft constraints should be satisfied if possible. We try to satisfy the stronger soft constraints in preference to the weaker ones. The paper cited above gives a formal description; here I just provide a brief description, mostly by way of example.

Every constraint should provide an *error function* that returns a number indicating how well the constraint is satisfied. The error should be 0 iff the constraint is satisfied. The error function can just have values 0 or 1 (depending on whether the constraint is satisfied or not), but for numeric constraints, for example on the position of some object, we usually want the error to be larger the further the current value is from a valid solution.

2.2 Different Options for Trading Off Unsatisfiable Soft Constraints

A key question is what we want to do if there are several soft constraints with the same strength that cannot be all satisfied. The paper defines a range of *comparators* that specify how they should be traded off. The three that will concern us here are *least-squares-better*, *weighted-sum-better*, and *locally-error-better*. Least-squares-better picks solutions that minimize the weighted sum of the squares of the errors of the constraints with a given strength. Weighted-sum-better picks solutions that minimize the weighted sum of the errors. Locally-error-better picks Pareto optimal solutions. Any least-squares-better solution is also a locally-error-better solution, and any weighted-sum better solution is also a locally-error-better solution.

To start off, here are some unsatisfiable soft constraints:

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{strong} & x = 8 \\ \text{medium} & x = 0 \\ \text{medium} & y = 0 \end{array}$$

The solution for all three comparators is $\{x \mapsto 8, y \mapsto 2\}$ — the strong constraint wins over the two medium ones.

If we can't satisfy a soft constraint exactly, we try to satisfy it as well as possible:

$$\begin{array}{rl} \text{required} & 10 \leq x \leq 20 \\ \text{medium} & x = 5 \end{array}$$

The error for $x = 5$ is $|x - 5|$, and so the solution is $\{x \mapsto 10\}$, for least-squares-better, weighted-sum-better, and locally-error-better.

Suppose we have two conflicting constraints with the same strength:

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{strong} & x = 0 \\ \text{strong} & y = 0 \end{array}$$

There are various ways of trading off constraints in this situation. There is a single least-squares-better solution, namely $\{x \mapsto 5, y \mapsto 5\}$, which has an error of $5^2 + 5^2 = 50$. This is less than the error for any other solution that satisfies the required constraint, for example $\{x \mapsto 6, y \mapsto 4\}$, which has an error of $4^2 + 6^2 = 52$, or $\{x \mapsto 10, y \mapsto 0\}$, which has an error of $0^2 + 10^2 = 100$.

There are an infinite number of weighted-sum-better solutions, namely all $x \in [0, 10]$, $y \in [0, 10]$ such that $x + y = 10$. So $\{x \mapsto 5, y \mapsto 5\}$, $\{x \mapsto 0, y \mapsto 10\}$, and $\{x \mapsto 7.8, y \mapsto 2.2\}$ are all weighted-sum-better solutions. However, Cassowary always finds a solution on the vertex of the feasible region, so in this case one of $\{x \mapsto 0, y \mapsto 10\}$ or $\{x \mapsto 10, y \mapsto 0\}$.

Least-squares-better and weighted-sum-better are so-called *global* comparators, since their comparisons involve all of the constraints. Locally-error-better is a *local* comparator that just works constraint-by-constraint, and finds a Pareto optimal solution. The locally-error-better solutions to the above example are the same as for weighted-sum-better. Here is an example where they are different:

$$\begin{array}{rl} \text{strong} & x = 0 \\ \text{strong} & x = 2 \\ \text{strong} & x = 10 \end{array}$$

There is just one weighted-sum-better solution (assuming the weights are all 1), namely $\{x \mapsto 2\}$. The error for $x = 2$ is $2 + 0 + 8 = 10$. It is larger for any other x – for example the error for $x = 0$ is $0 + 2 + 10 = 12$, and the error for $x = 7.5$ is $7.5 + 5.5 + 2.5 = 15.5$.

On the other hand, there are an infinite number of locally-error-better solutions: all $x \in [0, 10]$. So even though the error for both the $x = 0$ and $x = 2$ constraints is less for the solution $x = 2$ than for $x = 7.5$, $x = 7.5$ still does better on the $x = 10$ constraint and so it stays as a possibility.

2.3 Locally-Predicate-Better

In all of the previous examples, we used an error function that was larger the further the current value is from a valid solution. We could instead use a simple 0/1 error function. Weighted-sum-better with weights of 1 would become *unsatisfied-count-better* that just counts the number of unsatisfied constraints. Locally-error-better would become *locally-predicate-better*. DeltaBlue finds solutions that are both locally-error-better and locally-predicate-better (see Section 3.1).

2.4 Read-only Annotations

Another useful concept is that of a *read-only annotation* on a variable. A variable is read-only with respect to a given constraint if that constraint has no effect on the possible values for that variable. Read-only annotations thus shape the behavior of the constraint solver, but in the declarative theory presented in reference [6] the concept is captured by the theory rather than just being a property of the solvers. This is in contrast to the situation with the original ThingLab — at that time I didn't understand how to describe read-only annotations declaratively.

Read-only annotations are used in a variety of places in both the original ThingLab and in the reproduction. For example, a NumberPrinter thing, as used in the TemperatureConverter and SquareRoot examples, has a multi-way constraint relating the text in the display box to the number at the corresponding NumberNode: if the text is edited, the number is updated; and if the number changes to satisfy some other constraints, the text is updated. A Constant replaces this with a one-way constraint that make the text read-only. The user can edit the text and the number will be updated; but the system won't let you change the number and have the text update. Similarly, an Anchor has a one-way constraint between the anchor's position and the anchored point: you can move the anchor using the "move anchor" command, but otherwise the anchor isn't movable to satisfy other constraints.

Consider this example. (The ? on y is a read-only annotation, indicating that y is read-only with respect to the $x + 1 = y?$ constraint.)

$$\begin{array}{rl} \text{required} & x + 1 = y? \\ \text{strong} & x = 10 \\ \text{weak} & y = 5 \end{array}$$

The solution is $\{x \mapsto 4, y \mapsto 5\}$. Despite the strong $x = 10$ constraint, the read-only annotation on y in $x + 1 = y?$ means that the weaker $y = 5$ constraint wins.

Now consider the same example, but with the $x = 10$ constraint required.

$$\begin{array}{rl} \text{required} & x + 1 = y? \\ \text{required} & x = 10 \\ \text{weak} & y = 5 \end{array}$$

The theory says that this collection of constraints has no solution, and the solver will report that the constraints cannot be satisfied — even with $x = 10$ now being required, information can't flow back through the $x + 1 = y?$ constraint.

Here is an informal definition of what is a correct solution to a set of constraints that includes read-only annotations, quoted from [6]:

> One way of preventing a constraint from affecting the choice of values for a variable is to replace that occurrence of the variable by a constant. Thus, we begin the definition of the set of solutions to a constraint hierarchy H by forming a set Q of constraint hierarchies, where each element of Q is a constraint hierarchy with arbitrary domain elements substituted for the read-only variables. (Note that the same variable v may have read-only occurrences and normal occurrences. Only the read-only occurrences are replaced when forming elements of Q.) Intuitively, we guess a valuation for v, and then form a hierarchy using that guess. After making all possible guesses, we weed out solutions arising from incorrect ones. (Note that this is purely a specification of the meaning of read-only annotations, not a reasonable algorithm for actually solving such constraint hierarchies!) ... Next we solve the constraint hierarchies in Q, discarding any valuations that map the remaining unannotated occurrences of a variable to something different from what was substituted for its read-only occurrences. (In other words, we discard all valuations in which we guessed incorrectly.) This ensures that the permissible values for a variable won't be affected by read-only occurrences of that variable, but that they will be consistent with the read-only occurrences.

Let's work through the example. Here are a few of the infinite number of possible sets of constraints resulting in replacing $y?$ with various constants, and their solutions.

$$\begin{array}{rl} \text{required} & x + 1 = 3 \\ \text{strong} & x = 10 \\ \text{weak} & y = 5 \end{array}$$

Here we replaced $y?$ with 3, and the solution is $\{x \mapsto 2, y \mapsto 5\}$.

$$\begin{array}{rl} \text{required} & x + 1 = 4 \\ \text{strong} & x = 10 \\ \text{weak} & y = 5 \end{array}$$

This time we replaced $y?$ with 4, and the solution is $\{x \mapsto 3, y \mapsto 5\}$.

$$\begin{array}{rl} \text{required} & x + 1 = 5 \\ \text{strong} & x = 10 \\ \text{weak} & y = 5 \end{array}$$

Here we replaced $y?$ with 5, and the solution is $\{x \mapsto 4, y \mapsto 5\}$.

The only one of these that substitutes the correct value for $y?$ is the last one — thus deriving via the definition the solution that we identified informally earlier.

The constraint hierarchy theory gives a general definition of read-only annotations that allows cycles through read-only variables. There are examples in Section 3.1.4 of reference [6]. However, so far such cycles have never arisen in practical examples, and in the ThingLab replica we disallow them — if such a cycle is found, the system raises an exception that says the constraints are too difficult for it to solve.

2.5 Stay Constraints and Edit Constraints

Stay constraints are used to express the desire that things stay where they are if possible. For example, a weak constraint `stay` x says that (weakly) the value of x shouldn't change, unless there is some stronger constraint that says it should.

We formalize this as the constraint `weak` $x_t = x_{t-1}$? The subscript is used to denote values of x at different times. x_{t-1}? is read-only with respect to this constraint — we can't alter the past to satisfy constraints in the present. Stay constraints can have different strengths (including required).

For example, consider a line from p_1 to p_2, another point q, and a point-on-line constraint on the line and q. What should happen if we move p_2? Should the system move only q, only p_1, or both? We can express any of these options by using stay constraints with different priorities — if we prefer moving q over moving p_1, we'd put a stronger stay constraint on p_1.

Finally, an **edit constraint** is used to represent the common operation of repeatedly solving a collection of constraints with updated values for one or two of the variables, for example the x and y components of a point. Formally, this is just the constraints $x = 10$ and $y = 20$ (or whatever the new values are for x and y), but the solvers support incrementally resolving the constraints as new values for x and y are fed to it.

3 Constraint Solvers

Given a set of constraints, we want to find a solution for them. (In the interactive graphics context, if there are multiple solutions, generally we need to just find one, rather than all solutions.) In this section I briefly describe the three solvers implemented in the ThingLab reproduction. There are details of the relaxation implementation in the Appendix Section G, since it was implemented specifically for this reproduction.

3.1 DeltaBlue

A local propagation constraint has a set of methods for computing the values of each constrained variable that isn't annotated as read-only. For example, the constraint $x + y = z$ would have three methods:

$$x \leftarrow z - y$$
$$y \leftarrow z - x$$
$$z \leftarrow x + y$$

DeltaBlue [7, 9] is an efficient incremental algorithm for finding solutions to acyclic sets of local propagation constraints. It is incremental in two respects: in being able to rapidly re-solve a set of constraints given a plan that it has constructed and a stream of updated values for edit constraints; and in being able to incrementally update a constraint satisfaction plan when adding or removing a constraint. (A curious fact is that DeltaBlue is very widely used, but as a benchmark to test the performance and optimization capabilities of object-oriented languages and their virtual machines, rather than to actually find solutions to constraint problems.)

A key concept in DeltaBlue is that of *walkabout strength*; we'll use the walkabout strength found by DeltaBlue in the cooperating solvers architecture. The walkabout strength of a variable is the strength of the weakest constraint that can be flipped to being unsatisfied to accommodate a new value for that variable. Consider the following constraints:

required	$a + b = c_1$
medium	$a = 2$
strong	$b = 1$
required	$c_1 = c_2$
	$\dots$
required	$c_{49} = c_{50}$

The solution is $\{1 \mapsto 2, b \mapsto 1\, c_1 \mapsto 3, c_2 \mapsto 3, \ldots, c_{50} \mapsto 3\}$. The plan that DeltaBlue produces to satisfy the constraints involves first finding a, then b, then c_1, then c_2, and so forth, and finally c_{50}. The walkabout strength of c_{50} is medium, so that if we added a strong constraint that $c_{50} = 5$, DeltaBlue knows it can satisfy that constraint without having to examine the long chain of equalities. The new plan, after adding that strong constraint, would find $c_{50} = 5$, then $c_{49} \leftarrow c_{50}$, and finally $c_1 \leftarrow c_2$, $b \leftarrow 1$, and $a \leftarrow c_1 - b$. The $a = 2$ constraint loses.

An important limitation of DeltaBlue is that it only handles acyclic constraint graphs — it can't solve simultaneous equations. On the other hand, it can handle constraints over any domain for which local propagation methods can be supplied. This arises regularly in ThingLab examples for constraints relating a number and a string, where one of the methods takes the number and finds its print string, and the other method takes the string and parses it to find the number.

In use for interactive graphics, rather than a $c_{50} = 5$ constraint, we'd use an edit constraint so that fresh values for c_{50} could be fed to the plan and solutions found rapidly.

DeltaBlue can handle local propagation constraints that include methods that are not "unique state," that is, that can have multiple possible solutions. An example would be the constraint $x \geq 10$, which has an infinite number of solutions. In terms of the hard and soft constraint theory, the method for finding x actually takes that constraint, plus a weaker stay constraint that gives a specific value for x, and solves these constraints simultaneously. So if we have these constraints:

$$\begin{array}{rl} \text{required} & x \geq 10 \\ \text{weak} & x = 5 \end{array}$$

the solution would be $x = 10$. This is a bit *ad hoc*, and in the ThingLab reproduction, I don't use this aspect of DeltaBlue, and instead turn such constraints over to one of the other solvers.

So with this restriction, DeltaBlue finds solutions that are both locally-error-better and locally-predicate-better, since setting a variable using one of a local propagation constraint's methods will satisfy that constraint exactly.

3.2 Relaxation

As noted in the beginning of this section, both Sketchpad and ThingLab provided the **relaxation method**. This is an iterative numerical technique that relies on constraints providing an error function that returns 0 iff the constraint is satisfied. Briefly, when using relaxation, the algorithm considers each variable in turn. For each constraint on the variable, the algorithm finds a linear approximation to the error function by numerically computing its derivative at the variable's current value, then solving for a new value for the variable that minimizes the sum of the squares of the errors of the linear approximations of the constraints on the variable. For reasons of numerical stability, it works better not to try and move immediately to the solution, but rather move only partway. (Particularly if the variable needs to be moved by a large amount, the approximation may not be accurate enough.) The algorithm iterates until it converges. If all the required constraints can be satisfied, it returns an answer; if not, the algorithm says that no solution could be found. Relaxation finds least-squares-better solutions (approximately). If the algorithm says that no solution could be found, it's possible that in fact there is no solution that satisfies all the required constraints, but it's also possible that the algorithm got stuck in a local minimum even though in fact there is an answer. (Picky point: it's also possible that the algorithm finds an answer, but it's not a least-squares-better one because it got stuck in a local minimum. This hasn't been observed in practice however, only in contrived examples.)

Early in the project I asked Claude to assess several existing implementations of relaxation and compare that with implementing it from scratch; the recommendation was to implement from scratch, since there

wasn't that much code that would be reused from one of the existing implementations. Relaxation doesn't natively support different strengths of soft constraints, so in the implementation, the constraint errors are multiplied by 1000 for each stronger level (so 1 for weakest, 1^3 for weak, 1^6 for medium, and so on). The current version of relaxation is described in Section G of the appendix.

3.3 Cassowary

Cassowary [1] is an incremental version of the simplex algorithm that can solve sets of linear equality and inequality constraints. It can accommodate both hard (required) and soft constraints with different strengths, and finds weighted-sum-better solutions. It is optimized for interactive graphics, and has good support for handling edit and stay constraints and incrementality.

Cassowary is fast and accurate and can handle simultaneous equations and inequalities, but is limited to linear equations and inequalities over the reals. Relaxation is strictly more general than Cassowary, since it can handle nonlinear equations, but is much slower. The two algorithms trade off unsatisfiable soft constraints differently: relaxation finds least-squares-better solutions, while Cassowary find weighted-sum-better solutions.

Relaxation can also get stuck in local minima, but only for nonlinear equations — it would never get stuck on linear constraints; so that is a non-issue as far as a comparison goes.

4 Cooperating Solvers

The three solvers have different capabilities and performance characteristics, and there are many ThingLab examples that require using more than one solver in cooperation to find a solution (either DeltaBlue plus Cassowary, or DeltaBlue plus relaxation).

It turned out none of the existing cooperating solver architectures did entirely what I wanted, so I came up with a new one and had Claude implement it, given a detailed prompt. Here is an informal human-written overview followed by some examples; there is a longer Claude-written description in the appendix.

4.1 Cooperating Solvers Algorithm

The constraints are partitioned into *regions*: an arbitrary number of local propagation regions on the edges of the graph, each joined by a boundary variable to a central more complex region. Either Cassowary or relaxation is selected for the complex region (if non-empty), depending on the characteristics of its constraints: Cassowary if all the constraints are linear equalities or inequalities; relaxation if they are arbitrary constraints that have error functions; and otherwise the algorithm says that the constraints were too difficult for it to solve.

DeltaBlue is used to analyze the local propagation regions and solve each of them. Next, for each local propagation region, the cooperating solvers orchestrator adds to the central region a constraint that the local propagation region's boundary variable take on the value that DeltaBlue found for that variable, with that variable's walkabout strength as calculated by DeltaBlue. Then we use the appropriate solver for the central region (right now either Cassowary or relaxation). Then, for each boundary variable, if it still has the value that DeltaBlue found for it, leave it as is; if the value was changed by the central solver, run DeltaBlue to set that variable to its new value.

There are two special cases for the walkabout strengths of the boundary variables: required and weakest. The orchestrator treats those local propagation regions differently. This is purely an optimization — the algorithm would be correct without doing this — but in addition to efficiency, it's easier for a human to understand what is going on in those cases.

The first special case: if the walkabout strength is required, then use DeltaBlue to solve for the value of the variable, and add a required constraint to the central region that it take on that value. (This part is the same as for the general case.) That's it. (If the solver for the central region decided the variable should have some other value, that would just raise an unsatisfiable constraints exception.) This corresponds exactly to the propagation of known states one-pass method in the original ThingLab, rephrased for the hard and soft constraint theory.

The second special case: if the walkabout strength is weakest, leave that region alone initially, solve for the variables in the central region, and then use DeltaBlue to solve the local propagation region, setting the boundary variable to whatever the central region solver determined for it. This corresponds exactly to the propagation of degrees of freedom method in Sketchpad and the original ThingLab.

4.2 Cooperating Solvers Examples

Here are a few examples. These all use the same set of constraints, just with different strengths for the string constraints.

$$\begin{array}{rl} \text{required} & x + 1 = y + z \\ \text{required} & x + 1 = y - z \\ \text{required} & s_1 = \texttt{printstring}(x) \\ \text{required} & s_2 = \texttt{printstring}(y) \\ \text{medium} & s_1 = \texttt{"3"} \\ \text{strong} & s_2 = \texttt{"8"} \end{array}$$

This is a contrived example designed to require multiple solvers yet remain simple. The $x + 1 = y + z$ and $x + 1 = y - z$ constraints force these to be in a complex region that will be solved by Cassowary, since DeltaBlue can't handle the cycle. The two `printstring` constraints end up in separate local propagation regions. x and y are boundary variables. DeltaBlue determines the walkabout strength for x (medium) and its value (3); and the walkabout strength for y (strong) and its value (8). Then the orchestrator adds two constraints to the central region, so that we have:

$$\begin{array}{rl} \text{required} & x + 1 = y + z \\ \text{required} & x + 1 = y - z \\ \text{medium} & x = 3 \\ \text{strong} & y = 8 \end{array}$$

The solution for just the central region is $\{x \mapsto 7, y \mapsto 8, z \mapsto 0\}$. Cassowary found a different value for x than did DeltaBlue, so we flip the constraints in the first local propagation region. For the second local propagation region, Cassowary found the same value for y as did DeltaBlue, and so we leave that one alone. The final solution for the entire constraint graph is $\{x \mapsto 7, y \mapsto 8, z \mapsto 0, s_1 \mapsto \texttt{"7"}, s_2 = \texttt{"8"}\}$.

Now consider the same constraints, except that the final constraint is required rather than strong:

required	$x + 1 = y + z$
required	$x + 1 = y - z$
required	$s_1 =$ `printstring`(x)
required	$s_2 =$ `printstring`(y)
medium	$s_1 =$ `"3"`
required	$s_2 =$ `"8"`

The solution and solution process is the same as the previous example, except that since the walkabout strength of y is required, we don't need to consider whether Cassowary found the same value as did DeltaBlue.

Finally, we can make $x = 3$ weakest:

required	$x + 1 = y + z$
required	$x + 1 = y - z$
required	$s_1 =$ `printstring`(x)
required	$s_2 =$ `printstring`(y)
weakest	$s_1 =$ `"3"`
strong	$s_2 =$ `"8"`

This time we would hand the following constraints to Cassowary. (Note that we ignore the value for x from DeltaBlue, since that has the walkabout strength of weakest.)

required	$x + 1 = y + z$
required	$x + 1 = y - z$
strong	$y = 8$

The solution ends up the same as before (not surprisingly).

4.3 Correctness

Section E in the appendix is a fairly detailed (Claude-written) argument that the cooperating solvers architecture is correct, if we assume correctness for DeltaBlue, Cassowary, and relaxation. Briefly, if the constraints all fall into a local propagation region L, that region will be solved just by DeltaBlue, and by the assumption that DeltaBlue is correct, the overall answer will be correct. Similarly, if the constraints all fall into a single complex region C, that region will be solved by either Cassowary or relaxation, and by the assumption that these algorithms are both correct, the overall answer will again be correct. So now suppose there is a central complex region C and n local propagation regions $L_1 \ldots L_n$. Let $v_1 \ldots v_n$ be the corresponding boundary variables, $s_1 \ldots s_n$ be their walkabout strengths, and $x_1 \ldots x_n$ be the values found for them by DeltaBlue. We argue that the constraint $v_i = x_i$ with strength s_i is a valid stand-in for the entire region L_i for all $1 \leq i \leq n$ as far as the complex region is concerned: that is, when Cassowary or relaxation solves the complex region augmented with the stand-in constraints, the result will be either weighted-sum-better or least-squares-better (and hence locally-error better), and further, we can extend the solution for the complex region to a solution for the entire set of constraints and still have a locally-error-better solution.

However, all we can guarantee is that the overall solution will be a locally-error-better one; it may not be weighted-sum-better (if via Cassowary) or least-squares-better (if via relaxation). For example, suppose we have a local propagation region L_1 with 10 constraints that assert the boundary variable $v_1 = 3$ at medium strength, and another local propagation region L_2 with one constraint that the boundary variable $v_2 = 4$ at medium strength. Suppose also that the complex region just boils down to asserting that $v_1 = v_2$ at the

required strength. The solution to these constraints for both weighted-sum-better and least-squares-better is $v_1 = v_2 = 3$, but there is no guarantee the cooperating solvers algorithm will find that solution rather than $v_1 = v_2 = 4$ (both are locally-error better). This has not been a particular problem in practice.

4.4 Possible Extensions

This section describes several possibilities for augmenting the algorithm to support solving additional kinds of collections of constraints. I understand how to implement the first two extensions below — and the second extension in particular seems rather elegant — but unfortunately it's not clear that they would be particularly useful in practice. There are other extensions that would potentially be quite useful, but I don't understand how to implement them, at least yet.

First, it would be possible to extend the current system to handle some (but not all) cases with read-only variables in complex regions, rather than just saying the constraints are too difficult to solve. Suppose the complex region can be divided into two subregions R_1 and R_2, joined by a boundary variable with no read-only occurrences in R_1, and with all occurrences read-only in R_2. In this case, we can solve the new regions independently: first R_1 and then R_2. But we may be able to go further, perhaps carving off some local propagation regions from R_1 or R_2 or both, which might allow sets of constraints to be solved that can only be handled by local propagation and that were previously in the original complex region. There might also be additional read-only variables that allow further region splitting. The details get a bit complicated, but there don't seem to be any particular barriers to working through them and implementing this extension. However, I haven't seen any practical examples that would use this extension.

Second, another extension would be to support Cassowary/relaxation cooperation. One way to do this would be to allow linear regions hanging off a central complex region, in addition to local propagation regions. Capturing the effect of all the constraints in the linear region in the boundary variable becomes more complex than a single value and a walkabout strength as for local propagation regions. We would have a set of successively tighter bounds on the variables, each with its own strength. For example, the effect of a complicated set of linear constraints on a boundary variable x might have a required constraint that $x \leq 20$, a strong constraint that $3 \leq x \leq 10$, and a weakest constraint that $x = 5$. However, it's not clear how much benefit this would add over just handing all the constraints to relaxation, since relaxation can handle linear equalities and inequalities just fine (although Cassowary has much better performance).

A more useful direction for extensions would be supporting other kinds of solvers, but I haven't worked this part out. In particular, it would be useful to support conditional constraints and constraints over finite domains in addition to numeric constraints and local propagation constraints – an example would be different layouts selected according to some condition. Supporting this would probably involve adapting some of the theory and techniques devised by Molly Wilson for Hierarchical Constraint Logic Programming [12].

4.5 Comparison with Prior Work on Cooperating Solvers

Sketchpad included two solvers: relaxation, plus propagation of degrees of freedom. Propagation of degrees of freedom was a one-pass method, but in Sketchpad these constraints were still solved using their error functions rather than with local propagation methods. ThingLab provided relaxation, plus propagation of known states and propagation of degrees of freedom, both using local propagation methods for the constraints.

There is also a long history of other research on cooperating solvers, from the 1979 Nelson/Oppen work [8] to current solvers such as Z3 (`https://www.microsoft.com/en-us/research/project/z3-3/`). In work a decade or so ago on constraint reactive programming languages [4], I did use Z3 for some ThingLab-style examples, such as the bounded quadrilateral, but it isn't optimized for these sorts of applications.

In the ThingLab lineage, one cooperating solver architecture we implemented was Ultraviolet [5], which supports a more general architecture than is used in the ThingLab reproduction, but at the cost of needing a solver that can take the place of Cassowary or relaxation that is able to process soft constraints strength by strength (which neither Cassowary nor relaxation does natively).

References

[1] Greg J Badros, Alan Borning, and Peter J Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[2] Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[3] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[4] Alan Borning. Wallingford: Toward a constraint reactive programming language. In *Proceedings of the Constrained and Reactive Objects Workshop (CROW'16)*, Málaga, Spain, March 2016.

[5] Alan Borning and Bjorn Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints*, 3(1):9–32, April 1998.

[6] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[7] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[8] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[9] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[10] Ivan Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.

[11] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.

[12] Molly Ann Wilson. *Hierarchical Constraint Logic Programming*. PhD thesis, Department of Computer Science and Engineering, University of Washington, April 1993. Published as Department of Computer Science and Engineering Technical Report 93-05-01.

A Constraint and Constraint Solver Details

This appendix describes in some detail how the ThingLab reproduction represents constraints and how it satisfies them, and provides a correctness argument for the cooperating solvers architecture. The correctness argument makes use of the formal definitions from reference [6]. Unlike the rest of this paper (which was written entirely by a person), except for these introductory paragraphs, this appendix was written entirely by Claude, based on a number of chats. In the first chat, I gave Claude an outline of what I wanted, and in response it wrote the detailed description that is this appendix. In the second chat, I asked Claude to provide a correctness argument for the cooperating solver architecture, based on a suggested direction for that argument. Claude added this as Section E. Reading through it, I spotted a problem (not central to the overall claim, but a problem nonetheless). So the third chat started with asking Claude to confirm or refute my claim that there was a problem, and to repair the argument if there was a problem. Claude confirmed the existence of the problem, repaired it, and noticed an additional issue, which I asked it to repair as well. This turns out to be a non-issue, but Claude went down a deep rabbit hole before I realized that. Finally, there was yet another chat regarding repairing a small error in one of the definitions.

The rabbit hole, it turns out, was due to Claude not understanding what constitutes a correct solution to a set of constraints that includes read-only annotations. In the previous version of this note, I described read-only annotations in terms of what the solver was allowed to do, but only pointed to the theory paper for the formal definition. Claude assumed that correct solutions still needed to be locally-better if you just dropped the annotation — and when Claude asked, I didn't realize this was incorrect (it's now been over 30 years since we wrote the theory paper). The rabbit hole included Claude inventing a new concept ("mobility") in addition to walkabout strength, and algorithm changes that let it patch around the solver not having a local propagation method to update a variable via a read-only occurrence. This is now repaired in both the code and this paper, and I added a discussion of the theory behind read-only annotations to this document. Still, it was a revealing lesson in how deep a rabbit hole Claude could go down, with a very convincing-sounding explanation.

Now here is Claude:

The architecture separates three concerns. First, there is a solver-independent *theory layer*: constraint objects that say what a correct solution *is*, without any commitment to how one is found. Each constraint offers up to three *forms* — a linear form, a method form, and an error form — and a solver may claim a constraint only if it understands one of its forms. Second, there are three cooperating solvers: DeltaBlue (an incremental local-propagation planner), Cassowary (an incremental simplex solver for linear constraints, used via an existing library), and a relaxation solver written specifically for this project in the style of the original ThingLab and Sketchpad. Third, there is an orchestration layer: a *partitioner* that divides each compiled constraint set into regions, assigning each region to the cheapest solver that can handle it, and an *orchestrator* that runs the solvers over the partition, moving information between them in both directions.

The multi-form design is what lets several solvers share a single constraint vocabulary without knowing about one another, and it is also what lets the system degrade gracefully: a relation that can be stated linearly is available to all three solvers, a functional but nonlinear relation (a product) to two, and a relation that is nothing but a residual (the length of a line, the force in a beam) to relaxation alone.

B Constraints and Their Classes

B.1 Variables, Strengths, and Other Auxiliary Classes

A `Variable` is a named scalar holding a value — a number in the overwhelmingly common case, or a string (the contents of a text box, so that a constraint can relate a printer's displayed digits to its number). Variables do not normally store their values: the things in the picture remain the source of truth, and a variable harvested from a thing reads and writes through to the thing's primitive part. Solutions therefore land exactly where the user interface already looks, and the solver and the editors can never disagree about where a value lives. Boolean parts exist at the thing layer (a switch's `closed` flag) but are deliberately not solver values: they are read at compile time to choose which constraints a thing states, and are never bound as variables.

The `VariableRegistry` harvests variables from things, creating one `Variable` per (thing identity, field name) pair on first request. A fresh registry is built for every solve session, which is cheap at ThingLab's scale and immune to stale bindings after a delete or merge. Object identity does the aliasing work: merged points are the same `Point` object, so the shared endpoint of two joined lines harvests to a single x and a single y variable. Merging thus *replaces* equality constraints, as in the original ThingLab — no $x_1 = x_2$ constraints are ever generated for connected structure.

A `Strength` is one of a fixed, totally ordered set: `required`, then the four soft levels `strong`, `medium`, `weak`, and `weakest`. Required constraints must be satisfied — it is an error, reported to the user, if they cannot be — while soft constraints should be satisfied if possible, stronger ones in preference to weaker, following the constraint hierarchy theory. Strengths are singletons and compare by identity; `Strength.fromLabel` converts the editable string field a constraint thing carries (the user can retype `"required"` as `"weak"` in the structured editor) back into a strength, rejecting unknown labels.

Finally, a small solver-independent vocabulary spells out linear expressions: a `LinearTerm` is a coefficient–variable pair, a `LinearExpressionData` is a list of terms plus a constant, and a `LinearForm` is an expression related to zero by one of the operators `==`, `<=`, or `>=`. A helper, `expression()`, builds these from parts, combining duplicate variables by summing coefficients.

B.2 The `Constraint` Class and Its Three Forms

The abstract class `Constraint` carries a strength, an optional set of *read-only* variables (Section B.4), and three form methods, each of which defaults to null so that a constraint states only the forms it genuinely has:

```
abstract class Constraint {
  constructor(readonly strength: Strength,
              readonly readOnlyVariables: ReadonlySet<Variable>) { ... }
  abstract variables(): readonly Variable[];
  linearForm(bind?: LinearBinding): LinearForm | null { return null; }
  methodForm(): readonly MethodForm[] | null { return null; }
  errorForm(): ErrorForm | null { return null; }
  ...
}
```

The *linear form* is the expression-relation-zero spelling that Cassowary consumes. The optional *binding* argument supplies values for variables already determined elsewhere; a constraint may substitute bound

read-only variables into its constant term, or offer a linear form *only* when bound. The second case matters for relations that are only conditionally linear: a product $a \times b = c$ is linear in b and c whenever the binding answers for a, which is how Ohm's law with a known resistance becomes tractable for a linear solver (Section D.5).

The *method form* is what a local-propagation solver consumes: a list of satisfaction *units*, each unit a set of single-output methods, any one of which satisfies the unit by computing its output variable from the others. Methods are single-output and scalar throughout. A relation stated over whole points — one user-visible constraint, one strength — returns one unit per axis, so that DeltaBlue's single-output semantics hold per unit while the user still sees a single constraint. Units share their constraint's strength and read-only set, and a failure in any unit is reported as the whole constraint's failure, which is the honest answer: the user's constraint is the thing that is unsatisfied. A method reads its inputs through a reader supplied by the solver, because during propagation the values live in the solver rather than in the things.

The *error form* is what relaxation consumes: the scalar variables the error depends on, and a signed error function that is zero exactly when the constraint is satisfied. Errors are kept in commensurate model-space distance units — $|p_1 - p_2| - \mathit{length}$ rather than its square, the sine of an angle rather than a cross product — so that a single strength-weight table is sane across constraints. For inequalities the error is the hinge $\max(0, \mathit{violation})$: zero when satisfied, the violation's magnitude when not.

Each constraint also has two renderings: `describe()`, the solver-facing spelling of the compiled form, used in refusal messages; and `displayString()`, the browser-facing spelling shown by the structure format — the relation re-stated as a legal boolean expression over the parts' display names, with integer-scaled coefficients, read-only variables marked with a trailing `?`, and the strength appended. Some examples, as they appear in the browser:

```
line.end1.y == line.end2.y (required)
line.end1.x + line.end2.x == 2*midpoint.x (required)
p.x == xPosition? (required)
stay circle.center.x (weak)
length(line) == length? (required)
when closed: lead1.node.voltage == lead2.node.voltage (required)
sum(ends.map(end => end.xForce)) == 0 (required)
```

B.3 The Concrete Constraint Classes

LinearConstraint is the general-purpose class: a linear equality or inequality, expression-operator-zero. A number wire ($v_1 - v_2 = 0$), a horizontal line ($y_2 - y_1 = 0$), an adder ($a + b - \mathit{sum} = 0$), and an anchor's position pins are all LinearConstraints. The class says all three forms at once. The linear form is the expression itself, with bound read-only variables folded into the constant. The method form is derived automatically for equalities: for each variable v_i with nonzero coefficient c_i that is not read-only, the method $v_i = (-\mathit{constant} - \sum_{j \neq i} c_j v_j) \,/\, c_i$. Inequalities have no method form — local propagation cannot satisfy one — and read-only variables are never outputs. The error form is the expression's residual, hinged for inequalities. Stating a relation as a LinearConstraint rather than as a bare error form is precisely what makes it available to more than one solver.

MethodConstraint is the spelling for a relation that is functionally solvable but not linear: a product, a quotient, a number-to-string conversion. Its method units are supplied directly by the author. Its error form is likewise *supplied*, never reconstructed by inverting a method — the residual relaxation sees is a design decision. (The product and quotient constraints, for example, supply ϵ-guarded residuals and methods, so that a near-zero divisor produces an enormous but finite value the required check can still judge rather

than a NaN.) A MethodConstraint may also supply a *conditional* linear form — a factory that consults the binding and answers a linear form when enough of the variables are determined, or null when the relation is genuinely nonlinear as far as a linear solver is concerned. A MethodConstraint with no supplied error form is local-propagation-only; the number-to-string conversion that ties a printer's text to its number is such a constraint (its variables include a string, which relaxation cannot hold), and it is what makes the partitioner's "too difficult" refusal reachable at all.

ErrorFormConstraint is a constraint given directly by its error form: a label, the variables, the signed error function, and nothing else, so only relaxation can claim it. This is the right spelling for relations that genuinely have no other form: $|end_1 - end_2| - length$ is not linear, and mixes both axes of each point in one relation, so it has no method form either. The length constraints, the point-on-circle relation, the normalized parallel and perpendicular residuals, and Hooke's law in the bridge-truss things are all ErrorFormConstraints.

StayConstraint states $v = v_0$? — the desire that a variable keep the value it already has unless something stronger moves it, with v_0 captured at solver setup. Stays have linear and error forms but deliberately no method form: both DeltaBlue and relaxation treat stays natively (Section F). Explicit stays exist only where a thing asks for one; there is no stay-generation pass, because each solver supplies an *implicit weakest stay* on every otherwise-unconstrained variable in its own idiom — DeltaBlue variables start with weakest "walkabout strength," relaxation's warm start makes each variable prefer its current value, and the Cassowary adapter adds an explicit weakest stay for every variable not otherwise stayed. Things use non-required stays as a vocabulary for saying *who should move*: the midpoint constraint puts weak stays on its line's endpoints so that restoring the relation moves the midpoint rather than stretching the line, and the point-on-circle constraint puts weak stays on the circle's center so that the solver slides the point or resizes the circle before it moves the center.

EditConstraint represents the moving pointer during a drag: $v = input$?, read-only on the input, by default at strength `strong` so that a drag can never break a required constraint. An edit is a marker — a variable and a strength. Edits never go through `setup()`; they enter through each solver's incremental edit protocol (Section C), because the suggested value changes on every pointer motion.

BoundaryStay, a subclass of StayConstraint that lives with the solvers rather than in the theory layer, is created by the orchestrator to state a region-boundary value to a complex solver (Section D). Unlike a plain stay, whose v_0 is captured when the form is taken, its target is explicit and *mutable*: during a drag the orchestrator moves the targets of the boundary stays it inserted rather than rebuilding them. Recognizing a moving target is a published property of the type — relaxation reads the target afresh through an overridden error form, while the Cassowary adapter, whose underlying constraints are immutable once added, recognizes the class by name and states such a stay as an edit variable it re-suggests.

B.4 Read-Only Variables: Stating Direction Instead of Strength

Any constraint may mark some of its variables *read-only* (rendered with a trailing `?`): the constraint may read them but must never choose their values. Each solver honors the annotation in its own terms — relaxation never steps a read-only variable (reading it live at each evaluation), the derived method forms never offer a read-only variable as an output, and the Cassowary adapter folds every variable that any held constraint marks read-only into a constant at its current value.

The annotation turns out to be the right way to say many things that would otherwise be awkward strength contests. An anchor stores its own position and states $p.x = xPosition$? and $p.y = yPosition$?: the solver may place the point from the stored position and never the reverse, so a required anchor pins its point absolutely, a soft anchor springs back when a stronger pull releases, and the anchor itself moves only when something writes those fields (dragging it, or typing into the editor). A length constraint holds its `length` read-only, so dragging geometry moves endpoints and never quietly rewrites the length. A bounded point's

frame is read-only in every containment inequality, so the solver moves the point and never the walls. In each case the constraint states a *direction* once, instead of two constraints fighting at different strengths.

The solvers' own terms are not quite the annotation's meaning, and the difference matters in one case. DeltaBlue honors an annotation *per constraint*: a read-only variable is never an output of the constraint that marks it, and remains a candidate output for every other constraint. The two complex solvers honor it *per set*: a variable that any constraint in the region marks read-only is never chosen by the solver at all. The two readings agree exactly when nothing else in the region may write the variable — when each of its other occurrences in the region is itself read-only or belongs to a stay. (A stay asks only that its variable keep the value it has, which is precisely where the freeze holds it.) When some other constraint *could* write the variable, the freeze silently takes that constraint's say away, and the complex solver solves a more constrained problem than the one it was handed. Call a variable that is read-only in one constraint of a complex region and writable in another constraint of that region that is not a stay a *read-only conflict*.

The constraint hierarchy theory gives read-only annotations a general meaning that covers these sets too, cycles through read-only variables included (the examples of Section 3.1.4 of [6]), and a cycle is exactly what reaches a complex region, since DeltaBlue refuses to plan one. But such cycles have never arisen in a practical figure, and neither complex solver handles them. The partitioner therefore refuses a complex region holding a read-only conflict, with the same *too difficult* signal it uses for a region no solver can take, and both complex solvers refuse one at setup if a caller hands one to them directly. The rule is deliberately conservative: a complex region is solved simultaneously and has no direction, so the check does not try to tell a genuine cycle from an acyclic configuration that merely lands in one complex region. In practice that matters only with local propagation switched off, where everything is complex — an adder whose output feeds a one-way equality propagates with DeltaBlue enabled and is refused without it.

Only one of the shipped kinds tripped the rule, and the fix was to the kind rather than to the rule. A potentiometer's tap is held on its body by ties that are read-only on the body's location, so that dragging the tap slides it along the body and never tows the body. But the location was writable in the ties that keep the end terminals on the body's axis and the label beside it, and two ordinary configurations — a diagonal rotation, and a battery wired across both ends — put those ties in the same complex region as the tap ties. The potentiometer now holds its location read-only in every one of its ties, so the body is written only by its weak stays and by explicit moves; a terminal or the label can no longer tow it.

B.5 How Things State Constraints

Constraints do not live in a global store. Every thing answers the hook `constraints(registry)` with the (possibly empty) list of constraints it states over its own parts, harvesting variables from the registry it is handed. Compiling a composite is a walk over its parts — shared structure visited once — concatenating each thing's answer. For example, the entire compilation logic of the horizontal-line constraint kind is:

```
override constraints(registry: VariableRegistry): readonly Constraint[] {
  const p1y = registry.variableFor(this.line.end1, "y", "line.end1.y");
  const p2y = registry.variableFor(this.line.end2, "y", "line.end2.y");
  return [
    new LinearConstraint(expression([-1, p1y], p2y), "==",
                         this.compiledStrength()),
  ];
}
```

Two kinds of things state constraints. *Constraint kinds* are ordinary visible things — an anchor, a horizontal line, a midpoint line, a bounded rectangle — whose parts are (or merge with) the figure's own geometry and

whose constraints relate those parts; this is the original ThingLab's model of constraints as visible, insertable things. *Class constraints* are relations that come with a kind of thing: a number wire's equality, a resistor's Ohm's law, a beam's Hooke's law, a printer's number-to-string conversion.

Compiled constraints are *ephemeral*: they are rebuilt from scratch at every solve, against a fresh variable registry. This choice, extravagant-looking at first, buys a great deal of simplicity:

- There is no invalidation machinery. A merge or a delete simply changes what the next compilation produces.
- *Set constraints* — Kirchhoff's current law at an electrical node ($\sum leads.current = 0$ over however many leads the node has), force balance at a pin joint — expand to a single n-ary linear constraint over the membership *as it stands at compile time*. Connecting a circuit is merging nodes; the next compilation states the larger sum.
- A thing's constraint set may depend on its state. A switch compiles $v_1 = v_2$ when closed and $lead_1.current = 0$ when open — exactly one arm per solve — so flipping it is nothing but a recompilation, and the boolean flag is never a solver variable at all.
- Values may enter constraints as *coefficients*. A thermometer's scale bakes its `min`, `max`, and `height` fields into a linear constraint's coefficients; a rotated component's axis constraint bakes in the sine and cosine of its current rotation. Editing such a field just changes what the next compilation states. (A compilation that would be degenerate — an inverted scale — throws, which refuses the edit.)

The vocabulary in actual use spans linear equalities (position pins, axis ties, midpoints, adders, wires, batteries), products (multiply and divide operators, Ohm's law, a potentiometer's tap ratio), nonlinear geometry (lengths, circles, angle relations, Hooke's law), linear inequalities (a parameter confined to $[0, 1]$, containment of a point, line, rectangle, or circle within a frame), set constraints (current and force balance), string-valued display constraints (the number-to-string conversions behind printers, labels, and digital meter faces), and stays and edits.

C Triggering Constraint Satisfaction

There is no dependency tracking and no automatic re-solving: solving is triggered by user actions, each of which runs one *settle* or one *edit session*.

C.1 Settling

A settle is the batch case:

```
settle(root, extraConstraints?, settings?)
```

compiles the composite's constraints against a fresh registry, adds any *occasion constraints* contributed by the caller, and hands the whole set to the orchestrator, whose `setup()` immediately finds and writes back a solution — or throws. A thing with no constraints settles trivially; no solver is even built.

Settles run after every structural commit — inserting a thing, committing a constraint placement (whose merges attach the constraint to the figure), merging two things on a drop — and after every committed field edit. Occasion constraints express what the *gesture* means, over the same variable objects the figure

compiled: the canonical one is a *required stay on an edited value*. When the user types a number into the structured editor or a printer's text box, the commit settles with a required stay on the edited field; without it, relaxation would happily satisfy a network of number constraints by smearing the correction back over the value the user just typed. With it, the typed value joins the required set and propagates exactly — and an edit that genuinely contradicts the figure's required constraints is *refused*: the throw reaches the editor, which reverts the field, and the message strip reports `could not satisfy: ...` naming the constraint.

The refusal contract is uniform, and it rests on one invariant stated across all the solvers: *no write-back without a successful solve*. Every failing path — a required conflict in any solver, the partitioner's "too difficult" signal — restores every variable to its entry value before the error propagates. A refused commit therefore leaves the model exactly as the user placed it, plus an explanation.

C.2 Edit Sessions: Dragging

Dragging runs through an incremental protocol rather than repeated settles. On the *grab*, the tool compiles the composite's constraints, declares which variables the drag will edit (the partitioner uses this — Section D), and sets up the orchestrator, which settles once (the figure may snap to satisfaction if a stale state slipped in). It then opens one `strong` edit constraint per dragged scalar — an x, y pair per drag handle, or a single edit on a circle's radius for a circumference grab; a thing can override which variables its handles edit, which is how dragging an anchor edits the anchor's stored position. Because the edits are merely strong, required constraints always win: a drag can never break one.

Each *pointer move* suggests the offset-adjusted targets, and the solvers re-solve incrementally — exactly for the propagation and simplex regions, under an *anytime* 8-millisecond frame budget for relaxation, so the drag stays live while a large nonlinear figure visibly settles toward satisfaction, the classic relaxation feel. No required check runs mid-drag: a transiently violated required constraint during motion is part of that feel, not an error.

The *release* converges the final suggestion (so the drop point does not depend on how much processor time the drag happened to get), closes the edit episode, and runs a full settle with the required check, re-partitioned from the current values. If that settle refuses, the whole drag is undone: the tool restores the grab-time snapshot, which is always a satisfiable state. *Escape* restores the same snapshot — every variable, not just the dragged thing, since constrained neighbors moved too.

C.3 Incrementality

Incrementality operates at two levels. *Across* solves the system is deliberately non-incremental in structure: each settle recompiles and re-partitions from scratch, which is what lets merges, deletions, state-dependent constraint sets, and set constraints work with no bookkeeping. What makes this affordable is *numerical* incrementality: variables read and write through to the things, so every solve starts from the previous solution — a warm start — and a settle of an already-satisfied figure is nearly free.

Within an edit session everything is incremental: DeltaBlue extracts a plan once and re-executes it per frame at the cost of one method per constraint; Cassowary's incremental simplex re-optimizes the same tableau as edit values change; relaxation carries its state in the variables from frame to frame. DeltaBlue's incremental constraint addition and removal is also used by the orchestrator itself, to pin a region's boundary at a value the complex region chose and re-solve just that region.

D Cooperating Solvers: Partitioning into Regions

No single solver is right for everything. Local propagation is fast and exact but cannot handle simultaneous equations or inequalities; a simplex solver handles simultaneous linear systems and inequalities exactly but nothing nonlinear; relaxation handles arbitrary differentiable residuals but only approximately and iteratively. ThingLab therefore partitions every compiled constraint set and solves each part with the cheapest solver that understands it, with information crossing between the parts in both directions.

D.1 The Partition

The constraint set is viewed as a bipartite graph: a constraint touches the variables of its forms, and read-only touches count for connectivity, since a constraint that merely reads a variable is still coupled to whatever writes it. The partitioner splits the constraints, per connected component of this graph, into any number of *local-propagation regions* and at most one *complex region*:

- A constraint is *eligible* for local propagation iff it offers a method form or is a stay (which DeltaBlue takes natively). Inequalities, error-form-only constraints, and anything touching a string variable that lacks a method form are ineligible and seed the complex core.
- The eligible remainder falls into connected components, where connectivity runs only through variables the core does not touch; a variable shared with the core is a *frontier* of the component, not a connector.
- A component touching the core through more than one variable cannot be a local-propagation region (a region may have at most one boundary) and is absorbed into the core. A component with exactly one frontier becomes a candidate region with that variable as its *boundary*; with none, a free-standing region.
- Each candidate is trial-set-up in its own DeltaBlue solver. A cycle disqualifies it — simultaneous constraints are not local propagation — and so does a spurious required failure, because DeltaBlue's incremental satisfier is incomplete under *redundant* constraints (two required wires between the same two nodes form a two-cycle it reports as unsatisfiable, though the set is trivially satisfiable and relaxation solves it). A trial failure is therefore treated as a disqualification, never as a verdict: the component falls to the complex region, and a genuine required conflict is still refused — by the complex solver, at settle.
- Absorption is limited to constraints some enabled complex solver can actually take; the rest of an absorbed component re-partitions against the enlarged core, where a method-only constraint may yet earn a region of its own with a single boundary. Since each absorbing pass strictly grows the core, the loop terminates.
- Candidates sharing a boundary variable are merged — regions are maximal, so two regions can never share a boundary.

A counting argument shows why one complex region per component suffices: an eligible component lying *between* two complex parts has two frontiers, is therefore absorbed, and its absorption *connects* the parts. Two complex regions can only belong to disconnected figures, which exchange no information, so the pipeline below simply runs independently per component.

Each surviving region hands its trial solver onward, so the orchestrator needs no second setup — and, importantly, the trial declares no boundary inputs, because an input constraint would overwrite the very *walkabout strength* the orchestrator is about to read.

D.2 Choosing the Complex Region's Solver

The complex region goes *whole* to a single solver, by a three-tier priority rule. If every constraint in it offers a linear form over numeric variables — asked under the binding that folds the whole set's read-only variables, the same constants the adapter will freeze — the verdict is Cassowary, and the region is solved exactly, on every frame. Otherwise, if every constraint offers an error form, the verdict is relaxation. Otherwise the set is refused with the *too difficult* signal on the ordinary "could not satisfy" channel, a message that names why each constraint was refused local propagation (a cycle, a redundancy failure, two boundaries) and why each complex solver declined. A mixed region — some members linear-only, some error-form-only — falls to relaxation when relaxation can take all of it; splitting one region between two complex solvers is future work. One further condition applies once the choice is made: a region some solver would take is refused, with the same signal, if it holds a read-only conflict (Section B.4), the message naming the variable, the constraint that holds it read-only, and a constraint that could write it. The check comes after the choice so that a region no solver could take keeps its more fundamental message.

Which solvers may run at all is a session setting: three checkboxes, one per solver, consulted at each solve. Disabling DeltaBlue sends everything to the complex region; disabling Cassowary demotes linear regions to relaxation; disabling relaxation refuses the figures that need it, with the refusal naming the disabled solver — which is also the remedy. The setting exists to make the invisible choice visible: the same figure can be solved by different solvers and compared.

Notably, *which region is complex is a property of the gesture, not of the figure.* The partitioner is told which variables the upcoming drag will edit. A line carrying both a horizontal constraint and a midpoint constraint is pure local propagation at rest and under an endpoint drag; grab the *midpoint* and the y relations become simultaneous (the midpoint's y is given, so the tie and the midpoint relation jointly determine both endpoint y's), forming a complex region that Cassowary solves exactly — while the untroubled x relation stays DeltaBlue's.

D.3 The Pipeline: Information Flow Across Boundaries

With the partition in hand, the orchestrator runs a per-component pipeline. Free-standing regions just plan and execute — pure DeltaBlue, and when a component's complex region is empty *no complex solver is constructed at all*; a temperature-converter network settles entirely by propagation.

For a region with a boundary, the question is which side of the boundary informs the other. The answer is read off the region's DeltaBlue graph after setup, as the boundary variable's *walkabout strength* — the standard DeltaBlue bookkeeping, reused here as an interface between solvers:

- *Required* — propagation of known states. The region knows the boundary's value (a constant-fed chain): it executes its plan now, and the value enters the complex set as a required `BoundaryStay`. If the complex solver cannot satisfy it, the set is unsatisfiable.
- *Weakest* — propagation of degrees of freedom. The region is asking, not telling (a display node hanging off dragged geometry): the complex region solves first, and its value for the boundary is fed into the region as a required input, whose refreshed plan then runs.
- *In between* — a preference. The region executes and states the boundary at that strength; if the complex solver leaves the boundary there (within tolerance) the preference was conceded, and otherwise the boundary is pinned in the region at the complex solver's value — an incremental constraint addition, DeltaBlue's home ground — and the region re-solved.

The failure contract spans both solvers: the orchestrator snapshots every variable its constraints mention at entry and restores all of them on any failure before rethrowing. DeltaBlue regions hold their values in solver-internal variables until the whole pipeline has succeeded and only then write back — never echoing an input, since by write-back time the complex region may have moved a boundary.

D.4 Edit Routing

During a drag, each edit is routed to the region that owns its variable. Edits on complex-region variables go to the complex solver's own edit protocol; edits on a propagation region's variables become DeltaBlue input constraints with an extracted plan (per-move cost: one plan execution); an edit on a variable no constraint mentions gets the degenerate treatment, a direct write. Two refinements matter in practice: a boundary the pipeline fed *into* its region as a required input is determined by the complex region, so its edits route there; and an edited boundary a region holds only at intermediate strength is ceded to the complex region as well, with the region pinned to track the solved value per frame — otherwise a midpoint computed by a propagation region could sail past the wall that a bounded-point constraint in the complex region stopped its endpoint at.

Each pointer move then interleaves: run the edited propagation plans and write their values through; move the targets of the boundary stays the pipeline inserted; run the complex solver's suggest (anytime for relaxation, exact for Cassowary); and refresh the weakest-case boundaries back into their regions, so a display fed from dragged geometry updates live. The intermediate-strength re-pin check is deliberately skipped mid-drag — a transiently disagreeing boundary is the original relaxation feel — and the release settle, which re-partitions from the landed values, reconciles it.

D.5 Two Refinements: Edit Threats and Boundary Folding

Two refinements to the verdict deserve mention because they arise from the same underlying fact: both complex solvers treat whole-set read-only variables as constants, and Cassowary additionally bakes them into its tableau at setup.

Edit-threat demotion. Dragging a bounded thing's *frame* edits variables that are read-only in every containment constraint. A Cassowary-held region would clamp the bounded part against the walls as they stood at the grab, for the whole drag. A complex region any of whose read-only variables the upcoming edit drives therefore falls to relaxation for that session — relaxation's error forms read read-only variables live, so the advancing wall pushes the bounded part ahead of it frame by frame. And since neither complex solver will write a read-only variable however strong the edit, the orchestrator performs that write itself. With relaxation disabled the degraded path stands, documented: grab-time walls mid-drag, exact at the release settle. (Edits are session inputs, not constraints of the region, so a frame drag raises no read-only conflict: the edit drives the corners, and no *constraint* of the region may write them.)

Boundary folding. The pipeline already knows, from the walkabout strengths, which boundaries reach the complex region as determined values. The verdict may therefore *bind* those boundaries when asking whether a conditionally linear constraint could state a linear form, and the adapter folds them as constants. A required boundary (a fact — the region executes first and the value is written before the complex setup) folds freely. An intermediate-strength boundary (a preference) is folded only where the linearity actually needs it, and under a guard: after solving, every constraint at least as strong as the weakest folded preference must hold exactly, or the fold is rejected and the orchestrator rebuilds the region under the named fallback solver. Weakest boundaries are never folded — the complex region is being asked to determine them. The payoff is that a voltage divider, whose only nonlinearity is Ohm's law with resistances determined by their label chains, settles — and takes a resistance edit — under DeltaBlue and Cassowary alone, exactly.

E Correctness Argument for Cooperating Solvers Architecture

This section argues that the cooperating solvers architecture is correct in the following sense: whenever a settle returns a solution (rather than reporting that the constraints cannot be satisfied), the valuation it returns is a locally-error-better solution of the entire constraint hierarchy. The argument takes the three component solvers as correct: DeltaBlue returns locally-error-better (indeed locally-predicate-better) solutions of the constraint sets it is handed; Cassowary returns weighted-sum-better solutions; relaxation returns least-squares-better solutions; and every weighted-sum-better or least-squares-better solution is also locally-error-better. What remains to be argued — and it is a genuine question, not bookkeeping — is that being locally-error-better survives the *assembly*: the partition into regions, the summarizing of each local propagation region by a single stand-in constraint on its boundary variable, the complex solve over the abstracted problem, and the feeding of boundary values back into the regions. The argument is in English, but it is organized so that each load-bearing step is stated as a fact or lemma a formal proof would discharge; the closing remarks list what such a proof would have to add.

E.1 The Claim and Its Setting

Fix one settle. Let H be the hierarchy it solves: the compiled constraints of the figure — class constraints, the constraints of constraint things, stays, and any edit-commit stays — each at its strength and with its read-only annotations, together with an *implicit weakest stay* on every variable at its value at the start of the settle. Three points about this choice of H deserve emphasis before the claim is stated, because all three are load-bearing rather than technical.

First, the implicit stays are part of the hierarchy being solved, not an implementation convenience. All three solvers implement the convention: an undetermined DeltaBlue variable keeps its value and yields to anything, which is observably identical to a weakest stay; the Cassowary adapter states an explicit weakest stay for every otherwise-unstayed variable it mentions; and relaxation's warm start walks from current values to a nearby minimum, which we idealize the same way (of a piece with ignoring relaxation's finite convergence throughout this section). The convention matters to correctness at the weakest level. Consider a region whose boundary has weakest walkabout strength and which contains an unsatisfiable weakest-strength preference about the boundary's value, attached to a complex region that is indifferent to where that boundary sits. Propagation of degrees of freedom leaves the boundary wherever the complex solver put it and never pushes the region's weakest preference back — and without the stays that answer would be locally dominated, since moving the boundary toward the region's preference would improve one weakest constraint and worsen nothing. With the stays, "leave things where they are" is itself a weakest-level preference of equal rank, so declining to move the boundary for the sake of another weakest-level preference is a Pareto trade rather than a dominated choice. The stays are what make the architecture's oldest move — Sketchpad's propagation of degrees of freedom — sound under the hierarchy theory.

Second, solutions are the read-only-aware ones, in the theory's own sense. The formal treatment of read-only annotations is the fixed-point construction of the constraint hierarchy theory [6] (Section 3.1 there; Section 2.4 works an example through it): for a valuation θ, write $H\langle\theta\rangle$ for the hierarchy obtained by substituting θ's own value for every *read-only occurrence* of a variable — an ordinary hierarchy, with no annotations left — and call θ a solution of H exactly when θ is a solution of $H\langle\theta\rangle$. Two consequences carry weight below. A valuation is judged *in its own substituted hierarchy*, where its read-only occurrences are constants: so no competitor is ever credited with an improvement carried through a read-only occurrence — reducing a constraint's error by moving a variable that is read-only in it is not an improvement the theory recognizes, however the move might be justified elsewhere — and the errors of a read-only-bearing constraint are never compared across valuations that assign its read-only variables different values. (It is this definition under which the $x + 1 = y$? example earlier in the paper has no solution with $x = 10$ required, and, with

$x = 10$ merely strong, a solution that leaves the strong constraint unsatisfied.) The solvers respect the annotations by construction — no method, error term, or tableau column ever writes a variable to satisfy a constraint that holds it read-only — and several steps below dispose of hypothetical competitors on exactly this ground.

Third, the claim is about one settle's final valuation. Mid-drag frames deliberately make no such claim: relaxation's mid-drag suggests are anytime and carry no required check, and the intermediate-strength re-pin is skipped while the pointer is down. The release settle, which re-partitions and re-solves from the landed values, is a settle, and is covered.

Recall the comparator, in the form the argument uses. Call a valuation *admissible for* an ordinary hierarchy — one with no read-only annotations left — if it satisfies every required constraint of that hierarchy. For a strength level k (required is strongest, weakest is weakest), say that a valuation σ is *locally better* than a valuation θ *at level* k, within a given ordinary hierarchy, if: the error of every constraint at every level stronger than k is the same under σ as under θ; no constraint at level k has larger error under σ than under θ; and at least one constraint at level k has strictly smaller error under σ. A valuation θ is a *locally-error-better solution* of H if it is admissible for $H\langle\theta\rangle$ and no valuation admissible for $H\langle\theta\rangle$ is locally better than it at any level *in* $H\langle\theta\rangle$ — level-by-level Pareto optimality, judged in the solution's own substituted hierarchy, per the fixed-point construction above. The pool of competitors is therefore fixed by θ, not by the competitor: a competitor σ must satisfy the required constraints of $H\langle\theta\rangle$, with every read-only occurrence held at θ's value, and need not satisfy those of its own $H\langle\sigma\rangle$. (The two notions of admissibility can differ whenever σ and θ disagree on a variable that occurs read-only in a required constraint.) Throughout the argument, an unqualified "admissible" carries this meaning relative to the hierarchy in which the comparison is being made: $H\langle\theta\rangle$ itself, or its restriction to one component or to one local propagation region, or the complex region's abstracted problem C' defined below — in each case with the read-only occurrences held at the values of the valuation being judged. Note what the definition does *not* require: nothing at all is said about levels weaker than k; errors are compared constraint by constraint, never summed; and no comparison ever evaluates a read-only occurrence at two different values. All three features are used repeatedly below, and wherever the argument weighs a hypothetical competitor against a valuation, the comparison is made in the substituted hierarchy of the valuation being judged.

Theorem. If the orchestrator completes a settle of H and returns the valuation θ, then θ is a locally-error-better solution of H.

The assumptions, gathered in one place. (a) Each solver, handed a constraint set, returns a locally-error-better solution of exactly that set, satisfying its required constraints and respecting its read-only annotations, or else reports failure; DeltaBlue's solutions are also locally-predicate-better. (b) DeltaBlue's walkabout strengths correctly summarize its solutions, in the sense of the three Boundary Facts of Section E.3. (c) Local propagation methods are unique-state: executing a method sets its output to the unique value that satisfies its constraint, given the values of the other variables. (d) Arithmetic is exact; the tolerance used to test whether the complex solver left a boundary where its region put it is idealized to exact equality, as is relaxation's convergence. (e) The regions the partitioner hands DeltaBlue are acyclic, which the trial setup guarantees. (f) The abstracted problem handed to a complex solver (C', defined below) holds no read-only conflict (Section B.4): no variable is read-only in one of its constraints and writable in another that is not a stay. The partitioner guarantees this by refusing such regions.

Assumption (f) is what makes assumption (a) true of the two complex solvers as they are actually built. Both hold every variable that some constraint of their set marks read-only at its current value and solve for the rest, while (a) asks for a solution in the theory's sense — locally-error-better in its own substituted hierarchy. Let F be the frozen variables and θ the result, and suppose an admissible σ were locally better than θ at level k in $C'\langle\theta\rangle$. By (f), a variable $v \in F$ occurs there only as a constant (each read-only occurrence, substituted at $\theta(v)$) or in a stay. Every such stay asks for the value v held when the solver was set up, which is $\theta(v)$: a stand-in asks for its region's value, which the pipeline writes through first; an explicit stay captures

its variable at setup; and a boundary's implicit weakest stay is accounted to its region, not to C'. So no error in $C'\langle\theta\rangle$ depends on $\sigma(v)$ except those stays', which θ satisfies exactly. Resetting σ to θ on F therefore leaves it admissible (a required stay on v already forced $\sigma(v) = \theta(v)$) and still locally better than θ at k, since the only errors it changes become zero. But the reset valuation agrees with θ on F, so it competes in the very problem the solver solved, contrary to the solver's own guarantee there (weighted-sum- and least-squares-better solutions are locally-error-better, as the Remarks below note). Without (f) the argument fails at its first step: a non-stay constraint that may write v lets σ improve by moving v, which the frozen solver never considered — the defect the refusal exists to prevent.

Nothing is claimed in the other direction. When the architecture *refuses*, the constraints may nonetheless have had a solution: relaxation can stop in a local minimum, and the partitioner deliberately treats a DeltaBlue trial failure as a disqualification rather than a verdict for exactly this reason. The theorem says only that what is returned is never wrong.

E.2 Notation, and the Shape of the Computed Solution

Connected components of the constraint graph share no variables, and the pipeline runs independently per component, so components can be treated one at a time. Suppose some admissible σ were locally better than θ at level k. Then at least one constraint at level k has strictly smaller error under σ; choose one, and let it lie in some component. (σ may differ from θ, and may improve on it, in other components as well; that does no harm.) The spliced valuation that follows σ on that component's variables and θ everywhere else is still admissible, and still locally better than θ at k: on the chosen component its errors are σ's, which equal θ's at every level stronger than k, are nowhere larger at level k, and are strictly smaller for the chosen constraint; on every other component its errors are exactly θ's. A dominator of the whole would thus yield a dominator of a single component, so it suffices to show that θ restricted to each component is locally-error-better for that component's constraints. A component with no complex region is solved outright by DeltaBlue and is covered by assumption (a). So fix, for the rest of the argument, one component with a non-empty complex region.

Write C for the complex region's constraint set and $R_1, \ldots, R_n$ for the component's local propagation regions, region R_j sharing exactly one variable — its boundary b_j — with the rest of the component. (The partitioner absorbs any candidate region touching the core through two or more variables and merges regions sharing a boundary, so this shape is guaranteed; two regions share no variables with each other.) Each boundary variable's implicit weakest stay is accounted to its region, where DeltaBlue's walkabout bookkeeping already includes it.

The orchestrator computes θ in four steps. The two special cases — required and weakest walkabout strengths — are deferred to Section E.7, which shows each computes the same answer as this general scheme.

1. Each R_j is solved alone by DeltaBlue, yielding a valuation of the region's variables, a value v_j for b_j, and the walkabout strength w_j of b_j.
2. The complex solver is handed the *abstracted problem* $C' = C \cup \{S_1, \ldots, S_n\}$, where the *stand-in* S_j is the constraint $b_j = v_j$ at strength w_j, with error $|b_j - v_j|$ (a `BoundaryStay` in the implementation). It returns a valuation of the complex region's variables; write β_j for the value it gives b_j.
3. For each region with $\beta_j = v_j$ — the complex solver left the boundary where the region put it — the region's step-1 valuation is kept.
4. For each region with $\beta_j \neq v_j$, DeltaBlue re-solves R_j with an added required constraint $b_j = \beta_j$ (an incremental constraint addition, DeltaBlue's home ground), and that re-solved valuation is kept.

Step 4 always succeeds: it is reached only when w_j is weaker than required (a required stand-in is satisfied exactly by the complex solver, so $\beta_j = v_j$ and step 3 applies), and Boundary Fact 1 below says a region whose boundary has soft walkabout strength can accommodate any boundary value with all its required constraints still satisfied.

The returned θ is the union of the complex solver's valuation with the kept region valuations; these agree on every b_j by construction, so θ is well defined. It is admissible for $H\langle\theta\rangle$: every required constraint of H lies in C or in some R_j and was satisfied by the solver holding it, evaluated at θ's own values — which is exactly what satisfying its counterpart in $H\langle\theta\rangle$ means (the step-4 pin is a constraint of the re-solve, not of H). And every solver respected the read-only annotations of the constraints it held — none ever writes a variable to satisfy a constraint that holds it read-only — which is what the comparisons in $H\langle\theta\rangle$ below lean on.

E.3 What Walkabout Strength Certifies

Seen from the rest of the component, a region is visible only through its boundary variable, and the whole architecture rests on the walkabout strength w_j being an honest summary of the region's relationship to that variable. Recall its definition: the strength of the weakest constraint that can be left unsatisfied to accommodate a new value for the variable. The argument uses that summary in three forms. Each is stated relative to a *region solve*: either the step-1 solve of R_j alone or the step-4 re-solve with the boundary pinned — in both cases, DeltaBlue's valuation of the region's variables.

Boundary Fact 1 (the boundary can be moved, at cost w_j). For any value u, DeltaBlue can re-solve R_j with a required constraint $b_j = u$ added, and, when u differs from the value the previous solve gave b_j, the re-solve leaves unsatisfied, among the constraints the previous solve satisfied, one or more constraints of strength w_j, and possibly others at levels weaker than w_j (the implicit stays on the variables that moved among them); every previously satisfied constraint at any level stronger than w_j is still exactly satisfied, though possibly by a different method, save for numeric changes arriving solely through a constraint's read-only occurrences — changes the read-only semantics makes chargeless (the parenthetical below, and Section 2.4). In particular, when w_j is weaker than required the re-solve never fails, and every required constraint of R_j remains satisfied.

Boundary Fact 2 (it cannot be moved more cheaply). Let ρ be any admissible valuation of the region's variables giving b_j a different value than the region solve does. Then, relative to the region solve, ρ either changes the error of some region constraint at a level stronger than w_j, or strictly worsens the error of some region constraint at level w_j. Equivalently: there is no way to move the boundary while leaving every level stronger than w_j untouched and level w_j no worse.

Boundary Fact 3 (the region's stake in the boundary is worth at most w_j). No admissible valuation of the region's variables — *including* valuations that move the boundary — is locally better than the region solve at any level stronger than w_j. That is, at strengths stronger than w_j the region solve is already as good as the region can be made at any boundary value whatsoever, so the region has nothing stronger than w_j to say about where its boundary goes.

Facts 1 and 2 together say that moving the boundary costs one or more flips at level w_j but none at any stronger level; Fact 3 says that holding it is worth at most w_j to the region. All three are the operational content of the walkabout definition, and a formal proof would establish them about DeltaBlue by induction over the region's acyclic method graph. Fact 1 is DeltaBlue's ordinary incremental behavior: satisfying a required input revokes the weakest constraint on the reachable upstream chain — that chain's weakest strength is what walkabout strength *is* — and re-executes methods, which by unique-state methods (assumption (c)) leaves every re-planned constraint exactly satisfied. (The count at level w_j can exceed one: DeltaBlue revokes exactly one *enforced* constraint, but the previous solve may also have satisfied constraints it never enforced, and those lose their satisfaction as well when the re-executed methods move their variables. With $x = 5$ and

$y = 5$ both medium and $x = z$ and $y = z$ both required, only three of the four constraints can be enforced at once — four constraints, three variables — yet all four are satisfied at $x = y = z = 5$, and the walkabout strength of z is medium; pinning z at 10 revokes one medium constraint and leaves both unsatisfied. The implicit stays on the variables the re-execution moved are the weaker-level casualties. An enforced constraint at a stronger level survives the re-solve by re-executing its method. One satisfied *without* enforcement can lose its numeric satisfaction when the re-solve's changes reach it through a read-only occurrence; the read-only semantics prices that change at nothing — an error that appears only by way of a constraint's read-only occurrences is neither charged nor creditable, which is the point of the fixed-point definition (Section 2.4) — so the cost accounting above, the accounting Facts 2 and 3 refine, counts casualties at w_j and below only.) Fact 2 is the minimality of that chain: with the implicit stays in place every variable is the output of some satisfied constraint or stay, so a valuation that moves b_j must break something that was satisfied; and if only constraints weaker than w_j ever needed breaking, DeltaBlue's walkabout strength for b_j would have been that weaker strength. Fact 3 follows from local-predicate-betterness plus the read-only semantics: if some region constraint at a strength ℓ stronger than w_j could have its error reduced by moving the boundary at a price paid only at levels weaker than ℓ, then either DeltaBlue could have satisfied it through that channel — the channel's strongest obstacle costs w_j, which is weaker — and by locally-predicate-betterness it would have, contradicting the solve we have; or the constraint is insulated from the channel by a read-only annotation or a required conflict, and then the improvement does not exist in the terms the comparator uses: the comparison is made in the region solve's substituted hierarchy, where the constraint's read-only occurrences are constants and its writable variables are held by the constraints that determine them, so there is nothing left for a competitor to improve. Crediting an improvement carried through a read-only occurrence is precisely what the fixed-point definition never does — the worked example in Section 2.4 is this point in miniature.

E.4 The Stand-in as a Faithful Summary

Step 2 replaces each region by its stand-in S_j: $b_j = v_j$ at strength w_j. The Boundary Facts say why the substitution is faithful. From outside, the region's entire behavior is: left alone, it sets $b_j = v_j$ (step 1); overridden, it charges one or more flips at level w_j and nothing stronger (Facts 1 and 2); and it never pushes back at any level stronger than w_j (Fact 3). The single constraint $b_j = v_j$ at strength w_j presents the same level-by-level profile to the complex solver: satisfied, it holds the boundary at v_j; overridden, its error appears at level w_j and at no stronger level.

The substitution is a sound abstraction, not an equivalence. The stand-in's error $|b_j - v_j|$ is not numerically the error the region actually incurs when its boundary is forced elsewhere (those are the flipped constraints' errors, other functions of the imposed value), and one stand-in may summarize several strength-w_j preferences pulling in different directions, collapsing a family of Pareto-incomparable choices into a single preference. So the abstracted problem C' can have *fewer* locally-error-better solutions than the original component: the abstraction may narrow the architecture's choice among correct answers. Correctness needs only the other direction — no answer assembled from a locally-error-better solution of C' is locally dominated in the original hierarchy — and that is what the argument below establishes. Notice that the argument never compares error magnitudes across the substitution, only which level an error sits at and whether a given comparison moved it; that is exactly the information the stand-in preserves.

By assumption (a), the complex solver's step-2 valuation is a locally-error-better solution of C': Cassowary's weighted-sum-better and relaxation's least-squares-better solutions both are. One further observation is worth separating out because it disposes of a tempting shortcut. It is tempting to argue that $\beta_j \neq v_j$ implies that constraints *stronger* than w_j forced the boundary to its new value. That is not so: an equal-strength trade can do it — two strength-w_j demands on the complex side outvoting the stand-in under a weighted-sum objective, say — and Cassowary's vertex-picking or relaxation's averaging can select among equal-strength compromises in ways no single constraint forces. The argument therefore never asks *why* the complex solver

moved a boundary; everything below is stated for whatever value β_j it chose, forced or not.

E.5 Reattaching the Regions

Region Lemma. Let β_j be the final boundary value, and consider θ restricted to region j's variables — the step-3 valuation if the boundary was left alone, the step-4 re-solve if it was moved. That restriction is a locally-error-better solution of the hierarchy $R_j \cup \{P_j\}$, where the *pin* P_j is the constraint $b_j = \beta_j$ at strength w_j.

The pin is the region-side mirror image of the stand-in. The architecture summarized the region to the complex solver as one strength-w_j constraint on the boundary; symmetrically, when the region is judged on its own, the outside world's entire claim on the boundary enters as one strength-w_j constraint holding the value the complex solver chose. (This is the "at the walkabout strength" of the informal argument this section grew from.) Note that the kept valuation satisfies P_j exactly in both the step-3 and step-4 cases.

To prove the lemma, suppose some admissible valuation ρ of the region's variables is locally better than the kept valuation at some level ℓ in $R_j \cup \{P_j\}$, and take cases on whether ρ moves the boundary and on where ℓ sits relative to w_j.

If ρ gives b_j the value β_j: the pin's error is zero under both valuations and cancels from every comparison, so ρ would be locally better in R_j plus a *required* pin at β_j. For a step-4 region, that hierarchy is exactly what DeltaBlue solved, and assumption (a) refutes ρ. For a step-3 region, DeltaBlue solved R_j alone, and its solution already satisfies $b_j = \beta_j$ (here $\beta_j = v_j$); adding to a hierarchy a required constraint that a locally-error-better solution satisfies preserves its locally-error-betterness, since any dominator drawn from the smaller candidate pool is a dominator in the larger pool as well. Either way, contradiction.

If ρ moves b_j and ℓ is weaker than w_j: local betterness at ℓ requires every constraint at levels stronger than ℓ — the pin among them — to have exactly equal error. Equal error for the pin means zero, which means the boundary did not move. Contradiction.

If ρ moves b_j and $\ell = w_j$: the pin sits at level ℓ itself, and its error strictly worsens, from zero to something positive. A valuation that worsens any level-ℓ constraint is not locally better at ℓ. Contradiction.

If ρ moves b_j and ℓ is stronger than w_j: the pin sits at the weaker level w_j and constrains nothing at ℓ or above, so the strict improvement is at some constraint of R_j itself, with every region constraint at stronger levels untouched. This is a claim that the region can be improved at a level stronger than w_j, boundary free to move — exactly what Boundary Fact 3 rules out. Contradiction.

No such ρ exists, and the lemma holds. Notice what the lemma quietly absorbed: the complex solver may have moved the boundary for merely equal-strength reasons, or for no articulable reason at all (a vertex choice among ties), and the lemma is indifferent — the pin records the value, not the justification, and w_j is the right strength for it regardless.

E.6 Assembly: Nothing Admissible Does Locally Better

Now suppose, toward a contradiction, that some admissible valuation σ of the component's variables is locally better than θ at some level k: every constraint at levels stronger than k has equal error under σ and θ; none at level k is worse; and some constraint c^* at level k is strictly better. The constraint c^* belongs to exactly one of $C, R_1, \ldots, R_n$ — the stand-ins and pins are not constraints of H; they were devices of particular sub-solves. Three observations close every case.

First, σ moves no boundary whose walkabout strength is k or stronger. Suppose $\sigma(b_j) \neq \theta(b_j)$

with w_j at least as strong as k, and restrict attention to region j's constraints. Because σ is locally better at k, their errors under σ equal θ's at every level stronger than k — hence at every level stronger than w_j, and, when w_j is strictly stronger than k, at w_j itself — and are no worse at k. In every arrangement of w_j relative to k, then, σ moves the boundary while leaving region j untouched at all levels stronger than w_j and no worse at w_j: precisely what Boundary Fact 2 says no admissible valuation can do, relative to the kept region solve. (At w_j required this specializes to: no admissible valuation moves a required-walkabout boundary at all, since the disturbance Fact 2 forces would break a required constraint.)

Second, if c^* lies in the complex region, σ would improve the abstracted problem. Let σ_C be σ restricted to the complex region's variables, boundaries included, and compare it with the complex solver's valuation on C'. For the constraints of C: equal above k, no worse at k, and c^* strictly better — all inherited from σ's local betterness. For the stand-ins: every stand-in at level k or stronger has an unmoved boundary by the first observation, hence exactly equal error; stand-ins at levels weaker than k are irrelevant to a comparison at level k. And σ_C is admissible for C': the required constraints of C hold because σ satisfies them, and any required stand-in holds because its boundary is unmoved and θ satisfied it. So σ_C is locally better than the complex solver's answer at level k in C', contradicting assumption (a) for the complex solver.

Third, if c^* lies in region j, σ would improve the pinned region. Let σ_j be σ restricted to region j's variables, and compare it with θ's restriction in $R_j \cup \{P_j\}$, the Region Lemma's hierarchy. Region j's own constraints: equal above k, no worse at k, c^* strictly better. The pin: if σ left b_j unmoved, its error is unchanged at zero; if σ moved b_j, then by the first observation w_j is weaker than k, so the pin sits below the comparison level and is irrelevant. Either way σ_j is locally better at level k in $R_j \cup \{P_j\}$, contradicting the Region Lemma.

Every case ends in contradiction, so no admissible valuation is locally better than θ at any level, and θ is a locally-error-better solution of the component — and, by the composition observation, of H. This proves the theorem.

E.7 The Special Cases, and the Refinements

The orchestrator treats required and weakest walkabout strengths specially; the main text calls both treatments optimizations, and the argument can now say precisely why that is right.

Required. The general scheme would add the stand-in $b_j = v_j$ as a required constraint. The complex solver must then satisfy it exactly, so $\beta_j = v_j$ always, step 4 is never reached, and checking whether the complex solver agreed is vacuous — which is exactly the special-case behavior: solve the region, impose the value, done. Moreover the required stand-in excludes nothing: by Boundary Fact 2 at required strength, every admissible valuation of the component gives b_j the value v_j anyway, so imposing it can never turn a satisfiable problem into a refusal. This is propagation of known states, argued sound rather than assumed so.

Weakest. At weakest walkabout the region left alone holds the boundary at its stayed value — nothing in the region moves it — so v_j is the boundary's value at the start of the settle, and the stand-in the general scheme would add is $b_j = v_j$ at weakest strength: indistinguishable from the implicit weakest stay the boundary already carries in the complex solver's own problem. Omitting it hands the complex solver the same problem. The subsequent feed-back is step 4 with w_j weakest: the required input revokes exactly the boundary's stay and re-executes the region's plan. This is propagation of degrees of freedom, and — as noted in the setting — it is the implicit stays that make it locally-error-better rather than merely traditional.

Boundary folding (Section D.5) promotes selected stand-ins to constants so that a conditionally linear constraint can state a linear form. A folded required boundary is the required stand-in again — imposing a value every admissible valuation has anyway — and changes nothing. A folded intermediate-strength boundary

is a gamble guarded by the acceptance check: the fold is kept only if, after solving, every held constraint at least as strong as the weakest folded preference is satisfied *exactly*. When the check passes, the folded answer is in fact a locally-error-better solution of the unfolded C': at levels as strong as the weakest folded strength or stronger, every constraint's error is zero, so no competitor can improve there; and a competitor at any weaker level must hold all those levels exactly equal — in particular it cannot move a folded boundary, whose stand-in's zero error sits at one of them — so it is a competitor for the folded problem itself, which the folded solve's own locally-error-betterness (assumption (a)) refutes. The general argument then applies unchanged. When the check fails, the fold is rejected and the region re-solved without it under the named fallback solver, which is the general case. Folding changes the route, never the correctness of the destination.

Edit-threat demotion (also Section D.5) changes only *which* complex solver takes a region during a drag session. Both complex solvers return locally-error-better solutions of what they are handed, so the theorem is indifferent to the choice; the demotion exists for the fidelity of mid-drag frames, about which the theorem makes no claim, and the release settle it feeds into is an ordinary settle.

E.8 Remarks

Why locally-error-better is the right claim. The architecture mixes comparators: locally-predicate-better solutions inside the propagation regions, weighted-sum-better in a Cassowary complex region, least-squares-better in a relaxation one. The containments run one way, so locally-error-better is the strongest property all the pieces share. But something deeper is true: even if every piece used the same global comparator, the assembled answer would not in general satisfy that comparator globally, because the stand-in replaces a region's true error landscape with the surrogate $|b_j - v_j|$ — a weighted sum computed over C' and a weighted sum computed over the component are sums of different things. The local comparator never sums. It asks, constraint by constraint and level by level, whether anything improved without something at least as important being disturbed — and those are exactly the questions the Boundary Facts answer across the seam between a region and the complex core. Locality is what composes. (Within each piece the guarantee remains the stronger one: the regions' answers are locally-predicate-better for their pinned sub-hierarchies, and the complex region's answer is weighted-sum- or least-squares-better for the abstracted problem.)

Soundness, not completeness — and not uniqueness. The theorem is a soundness claim. Refusals may be conservative (relaxation's local minima; the deliberately coarse treatment of DeltaBlue trial failures; the refusal of read-only conflicts, assumption (f), which turns away some sets the theory solves), and among the typically many locally-error-better solutions, which one appears depends on the solvers' own selection behavior — the abstraction can narrow the choice, Cassowary lands on vertices where relaxation splits differences, and the accepted position is that handing a region to a different solver may change *which* correct answer appears, never whether the answer is correct. One robustness property falls out of the argument for free: nothing above uses the *maximality* of the propagation regions, only their shape — DeltaBlue-solvable, at most one boundary each. So the partitioner's policy of absorbing a candidate region into the complex core on any doubt is automatically safe; maximality is an efficiency property, not a correctness one.

What a formal proof would add. Chiefly four things. First, the fixed-point semantics of Section 3.1 of [6] carried through the argument's own machinery, together with the small lemmas the restriction steps lean on — that a restriction of a valuation to a region's variables can be judged in the corresponding restriction of its substituted hierarchy, with the boundary's value justified by the added pin or stand-in, and that the assembled θ is a solution of its own substituted hierarchy. Second, proofs of the three Boundary Facts from DeltaBlue's invariants, by induction over the acyclic method graph, using unique-state methods, the implicit stays, and DeltaBlue's rule that a constraint displaces only strictly weaker ones; Fact 3 is where the induction meets the read-only semantics, since the improvements it must exclude are exactly those carried through a one-way channel — improvements the substituted-hierarchy comparison never credits. Third, the bookkeeping for constraints left unsatisfied or unenforced by a region solve whose errors shift

when the boundary moves (their variables can sit on the flip channel); the domination hypothesis polices these at levels stronger than the comparison level, and the substituted-hierarchy comparison disposes of the shifts that arrive through read-only occurrences, but a formal proof must say so case by case. Fourth, the idealizations of assumption (d) made honest: with a tolerance in the agreement test and a finitely converged relaxation, the theorem becomes an approximate statement — locally-error-better up to the tolerances — whose precise form is a separate exercise.

F The Solvers and the API They Present

F.1 The Common Protocol

The orchestrator presents the same protocol surface as a single solver, so the rest of the system — the settle function, the dragging tool, the window's commit paths — constructs one solver object and neither knows nor cares how many solvers run underneath. The same protocol, named as an interface, is what either complex solver must present so the orchestrator can hold whichever one the verdict chose:

```
export interface ComplexSolver {
  setup(constraints: readonly Constraint[]): void;
  solve(): void;
  beginEdit(edits: readonly EditConstraint[]): void;
  suggest(values: ReadonlyMap<Variable, number>): void;
  settleEdit(values: ReadonlyMap<Variable, number>): void;
  endEdit(): void;
}
```

The contract: `setup` takes ownership of a constraint set — the region's constraints plus any boundary stays — and settles it immediately, writing back on success and throwing `UnsatisfiableError` (with every variable left or restored at its entry value) on a required conflict. `solve` re-settles, honoring boundary-stay targets moved since. The remaining four methods are the edit episode. The `settleEdit`/`suggest` distinction exists for relaxation, whose per-frame progress rides the wall clock and must land the release to convergence; Cassowary is exact per frame, so there the two are identical — the orchestrator calls one or the other by occasion, never by class. Deliberately absent from the protocol: warm-start promises (Cassowary's state is its tableau, so a caller that writes variables behind the solver's back must set up again — which the orchestrator does anyway, re-partitioning every settle), and any value read-out (both solvers write back themselves; the orchestrator reads boundary values from the propagation regions).

Both solvers must also recognize `BoundaryStay`'s mutable target, and both enforce the same required-residual tolerance (10^{-3} model units), because a required constraint's verdict must not depend on which solver happened to take the region.

F.2 DeltaBlue

DeltaBlue is an incremental planner for acyclic graphs of multi-way functional constraints, satisfying soft constraints locally-predicate-better; it is well described in the literature and not re-described here. The implementation is a direct TypeScript port of the classic implementations — John Maloney's Smalltalk original and Mario Wolczko's JavaScript translation, the `deltablue` of the standard benchmark suites — kept in the repository rather than taken from a package: the algorithm is some three hundred lines, no maintained TypeScript package existed, and porting reviewed, known-correct code preserved its behavior.

Two deliberate adaptations were made. Constraints are generalized from the original's unary/binary shapes to n-ary method lists; and a constraint's walkabout strength is computed over the *alternative methods' outputs* rather than over its inputs — the two coincide for the classic's reversible binary constraints, but a method-restricted constraint (one with read-only variables) must not advertise flexibility it does not have, or a later constraint will try to displace it along a route that is not there.

The surrounding system talks to an adapter, whose API is what the orchestrator expects of a local-propagation solver:

- `setup(constraints, inputs)` translates each constraint's method-form units into planner constraints (one per unit, carrying the parent's strength), installs stays natively — a stay is a method-less no-op whose output is its own variable, installed first and strongest first — and declares the region's boundary inputs last, as required input constraints. A required conflict surfaces as `UnsatisfiableError` naming the theory constraint; a cycle passes through untouched, being the partitioner's disqualification signal rather than a user-facing failure.
- `addInputs(variables)` pins additional boundaries incrementally (the contested-preference case above).
- `resolveInputs()` re-reads the input variables and runs the extracted input plan — one plan execution, the per-frame boundary refresh.
- `beginEdit` / `suggest` / `endEdit` run an edit episode as input constraints plus an extracted plan.
- `walkaboutOf(variable)` and `valueOf(variable)` are what the orchestrator reads: the boundary's walkabout strength that orders the pipeline, and a boundary value harvested without writing anything.
- `writeBack(excluding)` writes the session's values into the things — and nothing reaches the things before it. Values live in solver-internal variables for the whole session, which is what lets the orchestrator extend "no write-back without a successful solve" across all the solvers. Input variables are never written back, not even unchanged: by write-back time the complex region may have moved a boundary, and echoing the stale fed value would clobber the fresher solution.

F.3 Cassowary

Cassowary is an incremental simplex algorithm for constraint hierarchies over linear equalities and inequalities; it too is described elsewhere and not re-described here. The implementation used is `@lume/kiwi` (version 0.4.4 at the time of writing), the maintained TypeScript continuation of kiwi.js, itself a port of the C++ Kiwi rewrite of Cassowary. It was selected because it is pure TypeScript, dependency-free, and distributed as standard modules, so it drops into the project's toolchain with no configuration. The alternatives were assessed and declined: kiwi.js is the package `@lume/kiwi` forked from and superseded; cassowary.js — the sentimental choice, being the direct descendant of the original Smalltalk implementation — has been unmaintained for over a decade, predates module systems, and is slow beside Kiwi; C++ Kiwi compiled to WebAssembly adds a toolchain seam for no need at ThingLab's problem sizes; and writing a fresh implementation from the published description is solver-internals work the multi-form constraint design exists to avoid — and remains available later behind the same adapter surface.

The adapter translates between the theory layer and the library:

- Each theory variable mentioned by a stated form maps to one kiwi variable, seeded from its current value. Values live in the tableau during the session and reach the things only through write-back, which every successful solve performs.

- Every variable any held constraint marks read-only is folded to a constant at its current value — exactly the set relaxation refuses to step, so the two solvers freeze the same variables. The fold is faithful to the annotations because no held set contains a read-only conflict (Section B.4): the partitioner refuses such a region, and the adapter refuses one at setup.
- Each constraint states `linearForm(bind)` as one kiwi constraint, all three operators mapped. A constraint offering no linear form is a named setup error, parallel to relaxation's missing-error-form guard: silently dropping a constraint is the one failure a solver must never commit.
- The four upper strengths map onto kiwi's native strength bands; `weakest` is synthesized one more factor-of-10^3 rung down, matching the relaxation weight table rung for rung. Kiwi has no stay machinery, so compiled stays enter through their linear forms as ordinary soft constraints, and every other mentioned variable receives an implicit weakest stay at its current value, so no variable is left to the simplex's whim — the original ThingLab/Cassowary convention.
- A `BoundaryStay`'s moving target, and every edit constraint, becomes a kiwi *edit variable* at the appropriate strength, re-suggested per solve. (A required moving target is stated one rung below required, because kiwi rejects a required edit variable; the residual check enforces the requiredness.)
- Kiwi's refusal of an unsatisfiable required constraint is rethrown as `UnsatisfiableError` naming the theory constraint, and a required-residual check — against the same 10^{-3} tolerance as relaxation's — runs beside it, covering the two cases kiwi never sees: a required relation all of whose variables were folded to constants, and a required moving-target stay.

Two semantic differences from relaxation are accepted rather than patched. Kiwi's comparator is weighted-sum-better over numeric strength bands — an approximation of the constraint hierarchy, and the same approximation relaxation makes, with the same spacing. And kiwi minimizes weighted *absolute* error at a simplex vertex where relaxation minimizes weighted *squared* error: two equal-strength soft constraints fighting over one variable split the difference under relaxation and land at a vertex — one winning outright — under Cassowary. Handing a region to one solver or the other can therefore change *which* soft compromise appears, never whether the required constraints hold, and no test or behavior in the system depends on soft outcomes agreeing bit for bit.

G The Relaxation Solver

The relaxation solver was written specifically for this project, in the style of the original ThingLab and of Sketchpad before it: one-dimensional Gauss–Newton steps on individual variables over the constraints' error forms, with numerical derivatives, swept Gauss–Seidel fashion until quiescent. It is described here in full. Beyond the classic core, it acquired — each change motivated by a measured misbehavior — a required-constraint propagation pre-pass, a projection onto the required manifold, Aitken Δ^2 acceleration, a trust region, a cost-weighted distribution pass, hard boxes for required single-variable inequalities, and an anytime edit protocol.

G.1 The Error Function and the Weight Table

The solver minimizes

$$E \;=\; \sum_i w_i \, e_i^2,$$

one weighted term per constraint, where e_i is the constraint's signed error and w_i a weight geometric in strength: 10^6 for strong, 10^3 for medium, 1 for weak, 10^{-3} for a weakest-strength non-stay term — the

standard weighted-least-squares approximation of the lexicographic hierarchy. Weakest *stays* never enter E at all: the warm start (relaxation walks from current values to a nearby local minimum) is the implicit weakest stay.

Required constraints get the top weight, but two refinements matter. First, the required weight is re-scaled per constraint set to 10^3 times the strongest soft level actually present, rather than sitting at a fixed 10^9: required's exactness is enforced by the projection and the residual check described below, not by its weight, while a needlessly large weight makes Gauss–Seidel crawl — the per-sweep contraction along a stiff direction is roughly $w_{soft}/w_{required}$. Second, weight alone cannot distinguish "required, almost satisfied" from "required, unsatisfiable," so after every settle each required constraint's residual is checked against the tolerance (10^{-3} model units); a violation restores every stepped variable to its entry value and throws, with the message phrased "could not satisfy" — an iterative solver stuck in a local minimum is making a statement about this solve, not about the theory.

G.2 The Per-Variable Step

The basic move relaxes one variable v against the constraints incident on it (an incidence index built at setup keeps this from being quadratic). With each incident error e_i and its derivative e_i' with respect to v — estimated by centered differences with a probe scaled to the variable, $\epsilon = \max(10^{-4}, 10^{-6}\,|v|)$ — the step is the exact minimizer of the local quadratic model:

$$\Delta v \;=\; -\,\frac{\sum_i w_i\, e_i\, e_i'}{\sum_i w_i\, (e_i')^2}.$$

Three safeguards wrap it. The step is clamped to a *trust region*: no accepted step may move a variable by more than eight times the largest incident residual, so a vanishing gradient can no longer launch an unbounded $-e/g$ flight. (An edit's own residual — the distance to the pointer — counts toward the bound, so a legitimately long pull is never refused; the variable's own stay residual does not, since a stay measures how far the variable has already drifted, and counting it would let each amplified step license a larger one.) The step is then damped by a halving line search — up to eight attempts, each evaluated over the incident terms only — and accepted only if it improves the local weighted error by more than a *relative* quiescence threshold (10^{-12} of the local error), so that at rest, steps eventually round to zero improvement and nothing is written: no endless creep, no jitter. Finally, the value is clamped into the variable's box of hard bounds, described below.

G.3 Sweeps, Sweep Order, and the Distribution Pass

A solve is a sequence of Gauss–Seidel sweeps over the free variables — all variables mentioned by any term, minus every variable some constraint holds read-only (those are read live, never stepped; the solver refuses at setup a set in which a constraint other than a stay could write one) — until a sweep moves nothing, a sweep cap (1000) is reached, or, in the anytime case, an 8 ms frame budget expires mid-sweep, to be resumed next frame from where it stopped.

Sweep order is load-bearing. Variables are ordered *edited first, then slack, then stayed*, where the slack variables are those no soft term (stay or edit) cares about. Slack-first is what makes a required constraint satisfied by moving its slack before Gauss–Seidel gets a chance to crush a stayed variable and then crawl it back at $w_{soft}/w_{required}$ per sweep; edited-first, during a drag, makes each frame's fresh error appear before the slack variables' turn, so they absorb it and the weakly stayed variables see a zero residual at their turn and hold still. (Prevention is the whole fix: once a weakly stayed variable has drifted, single-variable relaxation cannot affordably walk it back.)

Between the slack steps and the stayed steps runs the *distribution pass*, which addresses the deepest problem of per-variable relaxation over sets with unstayed variables: sweep order alone used to decide who absorbed a constraint's residual, and the one-dimensional step is unbounded as its gradient vanishes. (Measured before the remedy: dragging a point constrained to a line at parameter $t = 0.99$ moved the line's far endpoint by one hundred times the pointer motion, and a parallel-lines constraint "solved" a 30-unit rotation by stretching an endpoint 400,000 units out — lengthening a line shrinks a normalized angular residual just as monotonically as rotating it.) In the pass, each relational constraint still carrying a residual e absorbs it by the cost-weighted minimum-norm correction over its free variables:

$$\Delta v_i \;=\; -\,e\,\frac{g_i/c_i}{\sum_j g_j^2/c_j},$$

where g_i is the constraint's gradient with respect to v_i and the motion cost c_i is the summed weight of the variable's incident stay terms, floored at the implicit-weakest weight (10^{-3}) for unstayed variables. This is the correction of least weighted squared motion $\sum_i c_i\,\Delta v_i^2$ that zeroes the linearized residual — van Overveld's step, shaped by the stated stays: weakly stayed variables move a little, anchored ones essentially not at all, and unstayed ones absorb almost everything. Who moves becomes a continuous function of the gradients and the stated stays rather than of registry order. Each component of the correction is clamped to the trust region and the variable's box, and the whole step passes the same line-search acceptance as every other.

Constraints coupled through a shared steppable variable are absorbed *jointly*, as one small least-squares system

$$\Delta v = C^{-1}J^{\mathsf{T}}y \qquad \text{with} \qquad (J\,C^{-1}J^{\mathsf{T}})\,y = -e,$$

where J is the group's numerical Jacobian, C the diagonal of motion costs, and e the residual vector; satisfied members of the group are included as zero-residual rows, so the correction to the violated members is made *within the satisfied members' null space*. (Measured without this: the two parametric equations of a point-on-line constraint, absorbed one at a time, traded their shared, cheap parameter back and forth and cancelled to no net progress.) A singular system — locally parallel constraints — falls back to sequential single-constraint absorption. Mid-drag, a stayed variable additionally defers its per-variable step while any incident relational residual remains above threshold: the distribution pass owns relational absorption, and the stayed steps are polish.

G.4 Acceleration: Aitken Extrapolation

Gauss–Seidel on a weighted hierarchy is stiff: it reaches the top level's manifold quickly, then crawls along it at a per-sweep contraction of about w_{low}/w_{high}. The crawl is geometric, so each variable's sweep-end trail is asymptotically a geometric sequence, and Aitken Δ^2 extrapolation can jump it to its limit. Because the slowest crawls are numerically invisible at short strides (the second difference drowns in rounding), attempts are made over strides that double — at sweep $3 \cdot 2^k$ the extrapolation looks back 2^k and $2 \cdot 2^k$ sweeps — until some stride sees the curvature. A jump is accepted only if it lowers the total weighted error, which keeps the extrapolation harmless where the geometric model does not hold; an accepted jump restarts the trail.

One refinement, in the settling case only: the extrapolated candidate is first projected onto the required manifold (with the projection described next) before being judged. When several variables crawl in coupled lockstep, the raw component-wise limits land *near* the answer but off the required manifold, and the required weight makes the comparison read that near-miss as a huge regression, rejecting every jump. During a drag the candidate is judged raw: the trail is then chasing a moving optimum, extrapolated "limits" are frequently large wrong-direction hops, and repairing them well enough to be accepted produces visible jitter.

G.5 Machinery for Required Constraints

Weight alone leaves the weighted optimum a residual of roughly (soft conflict) $\times$ $w_{soft}/w_{required}$ on each contested required constraint, and pure Gauss–Seidel is pathologically slow on some required networks. Three mechanisms bracket the sweeps.

A local-propagation pre-pass runs at the head of every settle — the original ThingLab's own first recourse of one-pass methods before relaxation, grafted onto error forms. A required constraint with a single free variable (a required stay) *determines* that variable; a required constraint all of whose free variables but one are already determined solves for the remaining one by damped one-dimensional Newton on its own error form — exact in one step for linear forms, a few steps for products. The wave runs to fixpoint, so a chain like the temperature-converter network resolves exactly, in either direction, before relaxation proper even starts. Satisfied *inequalities* never determine a variable — a satisfied hinge has no opinion about where in its half-space its variable sits. Where the wave stalls on a cycle — every remaining required constraint has two or more undetermined variables, yet the loop as a whole pins them — a simultaneous stage gathers each connected component of undetermined variables around an unsatisfied required constraint, together with every required constraint touching them and, when the component has at least as many equations as variables, solves it as one small damped Gauss–Newton system: numerical Jacobian, normal equations with a small Levenberg damping term, halving line search on the sum of squares. A stationary point that is not a solution (the saddle of $(x^2-3)^2$ at $x=0$, reachable when a squares network starts cold) is escaped once by a small deterministic perturbation. Underdetermined residues — multi-variable geometry, where the soft constraints rightly choose among solutions — are left to the sweeps.

A projection onto the required manifold follows the sweeps: up to fifty passes in which each required constraint moves its free variables by its own least-squares correction $\Delta v_i = -\,e\,g_i / \sum_j g_j^2$, displacing every variable a little instead of one variable a lot. The total motion is a fraction $w_{soft}/w_{required}$ of the figure's scale, so the soft comparator is disturbed by less than it already approximates. The projection maintains an active set: a variable pinned at a required bound, or held by a required single-variable constraint (a required stay), is dropped from other constraints' projection gradients — any correction sent to it is a share the pin immediately undoes, and leaving it in the gradient starves the variables that can actually move.

The residual check, described above, has the last word.

G.6 Inequalities

A *required* linear inequality over a single variable is read as a hard box on that variable: every stepped value, in every mechanism above, is clamped into its box. This is what keeps the weighted optimum from trading a required hinge away against a required equality with a steeper gradient — the hinge's own weight cannot protect it when the equality moves the shared variable a hundred times as fast, but a hard clamp can. Soft inequalities, and (in principle) multi-variable ones, remain pure hinge terms: a preference violated at a price. The multi-term containment constraints of the bounded things are handled through their hinge error forms whenever relaxation takes such a region.

G.7 The Edit Protocol

`beginEdit` registers one weighted term per edit at the edit's strength: $(v - \mathit{suggested})^2$, a strong term with a moving constant, initialized at the variable's current value. Each `suggest` updates the moving constants and re-relaxes under the frame budget — anytime, with no required check mid-drag. `settleEdit`, the last suggestion of a drag, writes the edited variables to their final targets outright and relaxes to convergence: landing the handles makes the relax's starting point canonical (edits exactly satisfied, everything else nearby)

rather than whatever mid-sweep state the last frame's budget truncation left, and if the targets conflict with a required constraint the relax pulls the edited variables back off them exactly as it would have anyway. `endEdit` removes the edit terms; the caller's release `solve()` then runs the full settle with the required check. Constraints can also be added and removed incrementally between solves, relaxation being trivially incremental: update the index, re-relax.

G.8 Scale and Known Caveats

The largest figure routinely solved is the shipped bridge truss: 93 constraints over 132 variables, 80 of them — the per-axis force balances and the nonlinear Hooke's-law members — in a single relaxation region. It settles from cold (unloaded geometry to full equilibrium) in about a second within the untouched sweep cap, with the largest required residual around 10^{-11} against the 10^{-3} tolerance, and from a warm start in a small fraction of that.

Two caveats are features of relaxation worth stating rather than bugs. Relaxation converges to the local minimum nearest its warm start: a squares network asked for $x^2 = 9$ answers 3 from a start near 3 and -3 from a start near -1, and a truss pushed through itself can settle to the buckled equilibrium. And value-space and pixel-space residuals share one weight table and one tolerance — commensurate at the demonstration magnitudes, and worth per-constraint scaling only if mixed systems misbehave. A related, measured property of the whole architecture: a relaxation-held figure never re-settles bit-identically (each further settle stops at a microscopically different point of a flat basin, moving values by around 10^{-10}), which is why the serialized bridge is held by its tests to six decimal places rather than byte-for-byte.

H The Division of Labor in Practice

A few worked figures show how the pieces combine.

A temperature converter. Constants, a multiply and an add operator, wires, and printers. Every relation offers a method form, so the partition consists entirely of free-standing propagation regions: DeltaBlue plans and executes, no complex solver is ever constructed, and typing 100 into the Celsius printer — a settle under a required stay on the typed digits — propagates to 212 exactly, or backward from a typed Fahrenheit value, the same network run the other way.

A level line with a midpoint, dragged by the midpoint. At rest, pure propagation. Under the drag, the y relations turn simultaneous and form a complex region whose constraints are all linear: Cassowary solves it exactly on the first frame and every frame after, while the x relation remains a propagation region. With Cassowary disabled, the same region falls to relaxation and lands the same drag to tolerance rather than exactly — visibly the same gesture, measurably different residuals.

A point held inside a box. The containment inequalities have no method form, so they always seed a complex region; being linear, it is ordinarily Cassowary's, and a dragged point stops dead at a wall, exactly, every frame. Dragging the *frame* edits read-only variables, so that session is demoted to relaxation, whose error forms read the advancing walls live and push the point ahead of them.

A bridge truss under load. Hooke's law has an error form and nothing else, so every member seeds the complex region and the verdict is relaxation — while the anchors' position pins and each weight's label conversion still partition off as propagation regions feeding required boundary values inward. Sag, tension, and compression are the settled answer to the load; retyping the weight's label re-settles the whole truss around the new value.

Refusals. Wiring together two constants holding different values, closing a switch across two points a required chain holds at different voltages, anchoring all four midpoints of a quadrilateral (whose midpoints form a parallelogram by construction) and dragging one — each is refused on the message strip with the offending constraint named, and the model restored to the state the gesture found. Disabling a solver a figure needs produces a refusal that names the disabled solver, which is also the remedy.