Constraints as a Design Pattern

Hesam Samimi Alessandro Warth Mahdi Eslamimehr Alan Borning
Communications Design Group, SAP Labs
Viewpoints Research Institute
Los Angeles, CA, USA
{hesam,alex,mahdi,alan.borning}@cdglabs.org

Abstract

Imperative programming has great merits. As the ubiquitous style, it is familiar, and its linear and step by step nature is favored by the human mind. Experienced programmers, however, are aware of its major flaw: it is easy for meanings to get lost in piles of code, making software hard to understand, extend, and debug. Constraint-based programming as an alternative has been observed to suffer much less from these flaws, where the "what" (the intention) is expressed rather than the "how" (the algorithm) in performing a computation. It is the job of the system to automatically achieve the intention through constraint solving. Sadly, poor performance and expressiveness has prevented this style from seeing widespread adoption.

We propose a general programming model as a kind of a sweet spot between imperative and constraint-based programming. Our aim is to leverage many benefits of constraint-based programming such as understandability, behavioral modularity, extensibility, etc., in a practical way and without suffering the breakdown of the approach as with the traditional constraint-based paradigm. This model enforces a certain organization where at the top-level a program is simply composed of a set of constraints. However, the constraints aren't necessarily solved by an external entity, and the programmer uses imperative code to specify (1) how each constraint should be solved in isolation, and (2) how to combine individual solutions.

We have implemented a tool called SKETCHPAD14 that incorporates this model in JavaScript, and built a number of realistic applications in it. In this paper we demonstrate the merits of our approach by comparing it with traditional imperative as well as constraint-based approaches.

Onward!'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3688-8/15/10...$15.00
http://dx.doi.org/10.1145/2814228.2814244

Categories and Subject Descriptors D.3.2 [*Language Constructs and Features*]: Constraints

Keywords Constraints, design pattern, programming languages, language design, declarative, mixed

1. Introduction

The conventional imperative programming style has great merits; humans tend to think in terms of step by step instructions, as in following a recipe to make a cake. Yet experienced programmers are well aware of when and how it disappoints and breaks down. Meanings and behaviors get buried down under piles of lines of code, making software hard to understand, explain, extend, reuse, and debug. These tasks, which make up a significant portion of software engineering, have been observed to be significantly simplified in the context of *declarative*, *constraint-based* programming.

Constraint programming allows stating "what" is desired from a computation and asking the system to automatically achieve it (*"I want a carrot cake that feeds 4 with less than 350 calories per serving!"*), rather than having to explain "how" it is achieved (*"Grandma's recipe says mix 2 cups of water with... and then..."*), often a more burdensome feat. The advantage of declarative programming over the imperative style is more than a matter of convenience. A declarative program is easier to understand, extend and modify, and so on. In the context of our cake example, it's easy to change the *what* to *"feeds 5 and 250 calories"* instead, but modifying grandma's recipe to account for that is not as simple!

There have been various success stories for constraints, for example, the use of constraint solvers for iOS and Macintosh auto-layout, constraints and solvers for planning and optimization, or finite domain constraints for model checking. However, using constraints and constraint solvers as part of the execution of software has by no means been widespread in mainstream programming. Those programmers who have engaged in some form of constraint-based programming must be aware of when and how it disappoints and breaks down! While some programs can be written elegantly, the obstacles of poor performance and expressiveness can surface rather quickly. While in the case of imperative code the problem becomes having to deal with entangled, hard to understand and debug, yet still functioning and usable code, for constraint-based programming this breakdown is catastrophic; the program cannot be written or executed after hitting the expressiveness or performance scalability walls. These challenges discourage most programmers away and often force them to fallback to imperative code.

This paper tells the story of how our curiosity in the very first constraint programming system–Sketchpad–eventually and accidentally led us to the following observation:

> *It is possible to leverage the benefits of both constraint-based programming (improved modularity, understandability, extensibility, emergent behaviors, etc.) and imperative programming (flexibility, power, practicality) together, by following a design pattern within an imperative language.*

Still operating in an imperative language, we allow an API to define constraints and use them as building blocks for making applications. The programming discipline requires that the program be organized in such a way that at the top level it is entirely described as a set of constraints. Those constraints, however, aren't necessarily solved by an external solver entity; they are simply solved by the programmer and using the imperative language itself. Thus imperative style computation such as assignments and looping constructs aren't part of the program's outside view, but rather used as part of the implementation of solving constraints.

Our approach is related to *mixed programming* paradigms, such as constraint-imperative programming [5, 7], where constraint solving is enabled within imperative code. In these languages an imperative (e.g., object-oriented) language is used for computations, while some of the dependencies are maintained automatically by declaring constraints. As a result, some of the aforementioned benefits of pure constraint programming are lost, since constraints are no longer the only players. On the other hand, we have found that our hybrid programming paradigm can be used to make realistic applications that enjoy those benefits of declarative programming (since the program is declarative from the top view) yet don't hit the limitations that often surface with constraint-based programming.

In this paper, we:

- propose a new discipline where imperative code is refactored into a set of constraint definitions.
- propose a concrete set of API and user interfaces (UIs) that implement the model and demonstrate its usage and benefits inherited from the declarative paradigms.
- present SKETCHPAD14, a publicly available implementation of the programming model for JavaScript that runs in the web browser.
- present our experiments implementing numerous applications in the system and share what we have learned in comparison to existing imperative and constraint-based approaches in the context of these applications.

In Sec. 2 we provide background on Sketchpad and a followup work which were the inspiration behind this paper. We then move on to introduce our proposed hybrid model of programming in Sec. 3, followed by an overview of the user-level language (Sec. 4), the definitions and expectations for the system's components (Sec. 5), the inside of the system (Sec. 6), and then from the standpoint of a developer (Sec. 7). Finally, Sec. 8 summarizes our experiences and evaluation of this work and Sec. 9 highlights the limitations.

2. Background

In 1961, before personal computers existed, Sutherland built a computer program called Sketchpad [16]. It was the first in many things: first graphical user interface, first computer drawing system, and so on. It is less commonly known that the tool is also the first object-oriented system, as well as first fully declarative programming system. That is, the user defines behavior for a drawing by simply expressing its constraints (requirements) (e.g. two line segments must be parallel) and the system will automatically and continually solve the constraints to ensure they are always maintained.

Many constraint-based programming systems came along after Sketchpad. Borning's ThingLab [3] generalized it by packaging it as a constraint-based "kit building kit," programmable within the Smalltalk environment. In later years, language researchers, realizing the scalability issues, seem to have veered towards mixing constraint-based programming with the common imperative style (e.g., constraint imperative programming [7]), keeping some but not all the benefits that purely declarative programming has to offer.

2.1 Constraint Solving in Sketchpad

An important innovation of Sketchpad was to employ a generic iterative numerical constraint solving approach called *relaxation* to let the system automatically maintain the constraints required by the user.

Constraints are expressed in terms of error functions returning a number indicating how far the state is from satisfying the constraint. The error functions for all constraints acting on each variable are approximated as linear equations. Often, multiple constraints refer to the same variable. Sketchpad's relaxation method uses the sum of squares for the error values of all constraints involving each variable. A *least-squares fit* is performed. The process iterates until the system converges. Because the system can, in many situations, smoothly and quickly converge to an acceptable solution, this form of constraint solving can be used interactively, as shown by Sketchpad and ThingLab. This relaxation method did not employ any domain-specific knowledge in solving a system of constraints—only generic numeric and iterative methods.

2.2 Thirty Years Later

In "30 Years after Sketchpad: Relaxation of Geometric Constraints Revisited," [18] van Overveld reworked Sketchpad's original relaxation algorithm for a small set of geometric constraints. Van Overveld proposed to let each type of constraint define its own *algorithm* to solve itself, in isolation. The algorithm is a function that returns a set of *deltas* (the amounts to change) for the variables it needs to update. For a better chance of convergence, the solution must be one that minimizes the total amount of change to the variables involved in the constraint. Unlike Sketchpad, which uses numeric differentiation and therefore does it automatically, in van Overveld's approach, the programmer must perform manual symbolic differentiation to come up with such algorithms separately for each constraint type.

Take the example constraint that maintains the length between two points *p*(x_1,y_1) and *q*(x_2,y_2) fixed at 1. Here is the algorithm to solve the `LengthConstraint` given points `p` and `q` for which the distance between is to be maintained at length `l`, written as a JavaScript (JS) function below. The returned value is a dictionary whose keys are the names of the objects whose properties are to be changed, and whose values are themselves dictionaries mapping property names to delta values.[1]

```
// delta fn for LengthConstraint:
function computeDeltas() {
  var deltaMagnitude =
    (distance(this.p, this.q) - this.l) / 2
  var delta = scaledBy(
    normalized(minus(this.q, this.p)),
      deltaMagnitude)
  return {p: delta, q: scaledBy(delta, -1)} }
```

To make things interesting, let us assume there is a second constraint acting on *p*(x_1,y_1), which says it should be fixed at the origin coordinates. The `computeDeltas` function for the `CoordinateConstraint` type simply returns the delta obtained from subtracting the target position from the current position of the point.

Once the deltas from all constraints (here the two above) are collected, they are averaged together, then the average is damped (divided by a constant) to increase the chances of convergence, and then applied to the variables. In a solution-converging situation (not always guaranteed with non-linear constraints), the system moves closer to a solution (i.e., reduces the sum of the error values) with each iteration, and usually converges quickly to an acceptable solution.

Next we introduce our new hybrid programming model inspired from van Overveld's approach.

[1] Note that auxiliary functions such as `minus` and `normalized` operate over and return `Point` types, which are themselves essentially JS objects with `x` and `y` properties. Also note that the function splits the delta evenly between the two points (i.e., it moves them equally in opposite directions), because this minimizes the sum of squares of the changes.

3. 53 Years Later: Sketchpad14

Firstly, Sketchpad, as well as van Overveld's tweak on its solving strategy, were quite limited on the set of constraint types handled and only suited for arithmetic constraints over real values. Nonetheless, they paint a beautiful picture of how elegant the design, implementation, explanation, and modification of a computer program can be. Each program is organized as a set of individual constraints that are declaratively stated. It's the job of the system to combine the results of all constraints, rather than leaving the user the often much more complicated task of defining what the aggregate behavior should be. Modifying and extending a program means simply removing and adding constraints. Dynamic interactions do the same. In our experiments, we have observed that this way of programming makes the whole process quite linear and digestible for the human mind.

Secondly, from one point of view, van Overveld's approach is inferior to Sketchpad's: the system has no real smarts in solving problems; the developer has to provide an algorithm that solves each constraint type beforehand. What we noticed, however, is that van Overveld's approach of delegating the responsibility for solving the individual constraint types from the system to the programmer is much more amenable to general programming.

Constraints as a Design Pattern in SKETCHPAD14

Van Overveld's alternative relaxation method for solving geometric constraints a la Sketchpad inspired us to build a fully general programming model around it. We call it SKETCHPAD14. Our intention was to go back and honor and implement many of the great aspects of Sutherland's Sketchpad, especially being fully declarative and enjoying all the benefits which that entails. Yet we wanted to give it a generality and practicality edge so that realistic, interactive applications can be developed in it.

As in the work that inspired it, the way to define and modify the behavior of a program in our tool is to add/remove constraints, rather than the common imperative programming practice of "adding or modifying some code." We take a pragmatic approach to constraint-based programming based on a generalization of van Overveld's approach. The programmer uses a general-purpose programming language to define the constraint and also to provide an *algorithm* to solve it. Note that there's nothing stopping that algorithm from simply invoking an external solver!

We built SKETCHPAD14 around a new programming model that we call *Constraints as a Design Pattern*, or CDP for short. We wanted to be able to run our programs in the web browser, so we built SKETCHPAD14 in JavaScript. Constructing a program in the CDP model involves describing four parts:

1. data and constraint **class** definitions,
2. **data**, as instances of data classes,

3. **continuous behaviors** given in terms of constraints that are always operating over the data[2], as instances of constraint classes,
4. **discrete behaviors**, that is, the definition of events. An event is defined by: (a) a *trigger*, e.g. an IO event, and (b) a *handler* method describing the changes that need to occur anytime the event is triggered.

The model restricts, though does not currently enforce, what can be done in a handler. It is primarily used to add/remove data or constraints to/from the store.

Thus the execution model is unlike the imperative style, i.e., statement by statement. At the top level, all behaviors are expressed in terms of constraints. Also the program is assumed to be reactive: as soon as constraints become unsatisfied the data is updated to re-establish them.

4. User Side: Using SKETCHPAD14 with Predefined Types

SKETCHPAD14 allows the user to define new data or constraint types. Yet the focus of this section will be on an end-user making use of existing definitions to build new programs. We will illustrate some of the benefits of our system, directly resulting from the fact that from the top view a program is purely declarative.

4.1 Example: CDG Logo

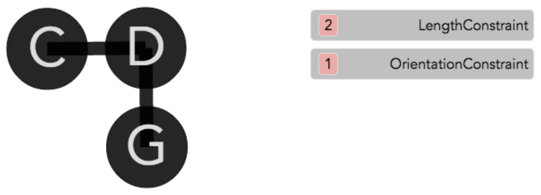


Figure 1. Logo example: viewing active constraints

Fig. 1 illustrates a program that displays our group (CDG)'s interim logo, which consists of three dots and two line segments. This is a reactive application: the user can drag one of these objects around. However, the structure of the logo needs to be maintained. To make this program, we instantiated three `Dots` (`p1, p2, p3`) and two `Lines` with the correct radius, text, color, etc. to match the figure.

The next step is to add the continuous behaviors. We will add two `LengthConstraints` to keep the lengths of lines fixed at $100px$ and one `OrientationConstraint` to keep the angle between them at $90°$, at all times:

```
addConstraint(new LengthConstraint(p1, p2, 100))
addConstraint(new LengthConstraint(p2, p3, 100))
addConstraint(new OrientationConstraint(
  p2, p1, p2, p3,  Math.PI / 2))
```

[2] Treatment of continuous constraints as real continuous values sampled at specific discrete times a la FRP [4] is subject of future work.

Finally, we need to define the reactive aspect of the program: if the user clicks on a dot, a `CoordinateConstraint` is added to require that it follows the mouse dragging. This constraint needs to be updated during the dragging:

```
registerEvent('mousedown', function(e) {
  if (e.pointedObject instanceof Dot)
    dragConstraint = addConstraint(
      new CoordinateConstraint(
        e.pointedObject.position,
        e.mousePosition)) })
registerEvent('mousemove', function(e) {
  if (dragConstraint !== undefined)
    dragConstraint.c = e.mousePosition })
```

The `registerEvent` call is a wrapper around JavaScript's `addEventListener` so that in the callback function SKETCHPAD14 objects can be accessed in relation to the event. Finally the drag constraint is removed on mouse up:

```
registerEvent('mouseup', function(e) {
  if (dragConstraint !== undefined) {
    removeConstraint(dragConstraint)
    dragConstraint = undefined } })
```

4.2 Exploring a Program's Behavior

In the previous figure, the set of active constraint types were shown at the top right of the screen, which helps program understandability. Although not shown, due to the registered events above one `CoordinateConstraint` temporarily appears during the user dragging event.

To quickly explore a constraint type, we can click on the number of instances (on the left) to get a list of individual constraints. Hovering over each will highlight the objects involved in the constraint (Fig. 2). To explore each constraint in more detail, we can click to get an *inspector* on it (Fig. 3). The class definitions of the constraint type itself can be viewed by clicking on the name of the constraint type (rather than the instances) in the right pane. We'll see examples of those definitions in the next section.

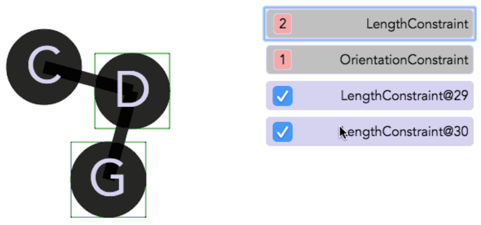
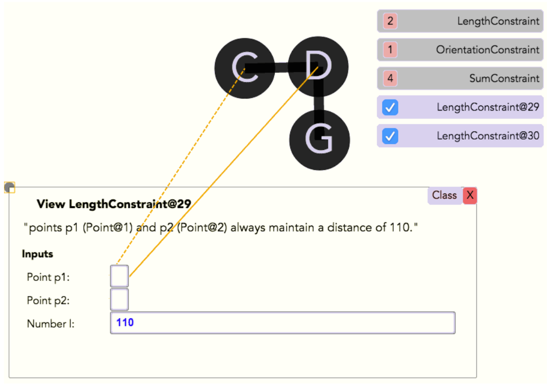


Figure 2. Viewing involved objects in a constraint

The inspector displays the properties belonging to its associated object. For those properties which are objects, arrows are drawn to the objects in the scene. Note that properties of objects are typed.

We can modify the properties of the inspected object (here the constraint). For example, in Fig. 4 we had modified the required angle between the two lines segments. We can also modify the object properties of the inspected entity. For



View LengthConstraint@29

"points p1 (Point@1) and p2 (Point@2) always maintain a distance of 110."

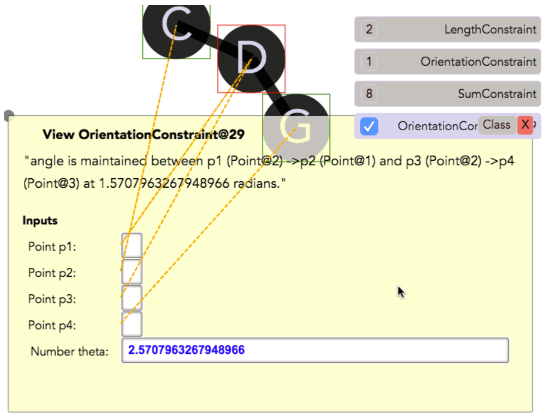
Inputs

Point p1:

Point p2:

Number l: 110

Figure 3. Inspecting a constraint & constraint description



View OrientationConstraint@29

"angle is maintained between p1 (Point@2) ->p2 (Point@1) and p3 (Point@2) ->p4 (Point@3) at 1.5707963267948966 radians."

Inputs

Point p1:

Point p2:

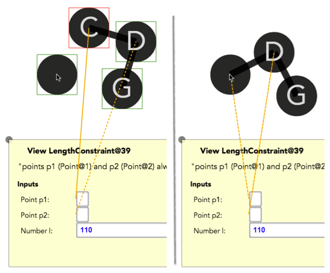
Point p3:

Point p4:

Number theta: 2.5707963267948966

Figure 4. Modifying a primitive value in a constraint

example, in Fig. 5 we modify one of the length constraints by changing one of the dots it operates on. Notice that when each listed property is clicked on, all compatible values (of the same type) in the scene are highlighted. By clicking on one of them we can change the property to reference the clicked object. Here we chose to use a concept in Sketchpad called *merging*, where the target object inherits all behaviors of the source object (recursively in its object properties) and then the source object is removed (See Fig. 5).



View LengthConstraint@39

"points p1 (Point@1) and p2 (Point@2) alw

Inputs

Point p1:

Point p2:

Number l: 110

View LengthConstraint@39

"points p1 (Point@1) and p2 (Point@2

Inputs

Point p1:

Point p2:

Number l: 110

Figure 5. Modifying an object property (via *merging*) in a constraint using the inspector: (*left*) before (*right*) after

To explore and understand a program and its behaviors, we may wish to turn on and off constraints to see the change in behavior, as shown in Fig. 6. Note that doing the same in an imperative program is likely to be not as simple.

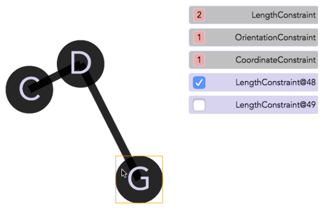
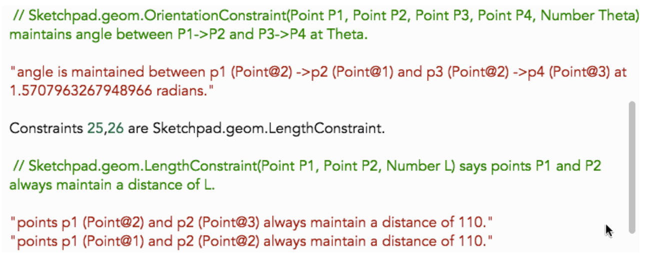


Figure 6. Mixing & matching behaviors: a `LengthConstraint` disabled

Another feature is an English sentence summarizing the behavior. For the constraint inspected in Fig. 3 it reads *"dots p1 & p2 always maintain a distance of 110."* SKETCHPAD14 can also generate a whole English description of the program. This involves giving a parameterized description of each involved constraint type, as well as a description of individual active constraints as the one above, along with a listing of existing objects and their states, and finally the set of registered events. Fig. 7 shows part of this description. In Sec. 7 we examine how these are generated.



```
 // Sketchpad.geom.OrientationConstraint(Point P1, Point P2, Point P3, Point P4, Number Theta)
maintains angle between P1->P2 and P3->P4 at Theta.

"angle is maintained between p1 (Point@2) ->p2 (Point@1) and p3 (Point@2) ->p4 (Point@3) at
1.5707963267948966 radians."

Constraints 25,26 are Sketchpad.geom.LengthConstraint.

 // Sketchpad.geom.LengthConstraint(Point P1, Point P2, Number L) says points P1 and P2
always maintain a distance of L.

"points p1 (Point@2) and p2 (Point@3) always maintain a distance of 110."
"points p1 (Point@1) and p2 (Point@2) always maintain a distance of 110."
```

Figure 7. Viewing a program description

5. Definitions

To get started explaining the details of the proposed programming model, let us discuss some definitions and expectations from the programmer as well as from the system.

In its full generality, constraint solving over a set of variables in CDP occurs at three levels. At the bottom level, individual constraints are solved in isolation. At the next level, a *merge* logic combines the solutions from all the constraints on a given variable to find one consolidated solution for its updated value. At the top level, another logic coordinates which collected solutions and in what order should be applied and how to proceed. In the case of the van Overveld algorithm, for example, this involves finding all the new values for the variables, simultaneously updating them, and iterating until a fixed point is reached.

5.1 Level 1: Individual Constraints

Constraints are relations over a subset of variables in the program that we want to hold. They are implemented in CDP in terms of classes whose instance variables are the variables they operate on. They define the following functions, which are potentially functions of the current *pseudo-time* of the system, so that constraints can be temporal.

`types` gives a type to the variables, enabling run-time checks such as the type-oriented UI features shown earlier.

`predicate` is a boolean function, returning `true` iff the constraint is satisfied.

`error` returns a numeric error value as an indicator for how far the state is from a solution. It returns 0 (or close to 0) iff the constraint is satisfied.[3]

`epsilon` defines an upper bound ϵ for an acceptably small error value.

`solve` takes the names of a subset of variables constrained by this constraint (ones allowed to change, i.e., not read-only), and returns a list of *patches* (new values for those variables) to make the constraint hold. The returned solution is represented as a dictionary where the keys are the names of the object instance variables belonging to the constraint (for simplicity we only allow constraints over the properties of objects, e.g., in the `LengthConstraint` they are called `p1` and `p2`) and the values are themselves dictionaries, mapping the name of a property belonging to an object to a new value.

The patches should be such that, if we apply them, the constraint will be satisfied. In addition, for the van Overveld-style algorithm used in Sketchpad14, the new values should minimize the sum of the squares of the changes to the variables. (The goal here is to perturb the system as little as possible when solving the constraints.)

Here is their definition for an instance of `LengthConstraint`:

```
LengthConstraint.prototype.propertyTypes =
  {p1: 'Point', p2: 'Point', l: 'Number'}

LengthConstraint.prototype.error =
  function(pseudoTime) {
    return distance(this.p1, this.p2) - this.l }

LengthConstraint.prototype.solve =
  function(pseudoTime) {
    var p1 = this.p1, p2 = this.p2
    var deltaMagnitude =
      (distance(p1, p2) - this.l) / 2
    var delta = scaledBy(
      normalized(minus(p2, p1)), deltaMagnitude)
    return {p1: plus(p1, delta),
      p2: plus(p2, scaledBy(delta, -1))} }
```

[3] When the `error` function isn't provided, the system assumes an error value of 0 when satisfied and 1 when if not. Similarly, if the `predicate` function isn't provided, the system uses the `error` and `epsilon` functions to determine whether or not the constraint is satisfied.

Contrast the above with the `computeDeltas` function for the `LengthConstraint` in the van Overveld approach, given previously. There this function was used to both express and solve the constraint and it returned a set of deltas. Here, on the other hand, we have separated the expression and solving of the constraint, and the solution is the set of new values the constraint is proposing for the coordinates of the two points. It so happens that these new values are obtained by adding delta values to the current values. Yet this logic was left as a choice to the implementer of the constraint type.

5.2 Solution

If we only had required constraints, and no other considerations, the desired properties of a global solution would be obvious: we need to satisfy all the constraints. However, this is not the only consideration. First, we don't want solutions that are wildly different from the starting state. Second, we might want to have explicit soft (preferred but not absolutely required) constraints, perhaps with multiple priorities, in addition to the required constraints. To accommodate this, we use the definition of a solution to a collection of hard and soft constraints given in [2]. For relaxation-type solvers, this involves minimizing the sum of the squares of the errors for the highest-priority soft constraints, then if there is still some freedom left, the sum of the squares of the errors for the next priority, and so forth.

5.3 Level 2: Merge Function

We take the approach of van Overveld to collect the solutions from unsatisfied constraints and *merge* them in order to come up with one consolidated solution. Unlike the van Overveld approach, however, in CDP we do not use a hardcoded merge function, but rather let the programmer define it:

`merge` given the current value of a variable (`curr`) and the set of collected new values (`sols`) obtained from individual constraints that wanted to change it, this function decides how to consolidate the set of values into one new value.

We let the programmer designate different merge functions to different variables. A particular merge method is specific to a constraint satisfaction algorithm. A general property for any merge function is that applying it must result in an equally good or better solution to the constraints overall (where "good" is again as defined in [2]). For iterative numeric techniques, such as relaxation, this means that the sum of squares of the errors for the affected constraints (or all constraints, same thing) must remain the same or decrease; this means that the algorithm does hill climbing.

For example, to perform the van Overveld style averaging and damping the deltas we can define:

```
dampedAverageMergeFn = function(curr, sols) {
  var damping = 0.25, sum = 0
  sols.forEach(function(v) { sum += v })
  var avg = sum / sols.length
  return curr + (damping * (avg - curr)) }
```

As another example, we may use a merge function for finite domain constraints, where each constraint restricts the possible values that the variable can take on. Then, if one constraint says that the value of the variable must be in the set `{red, yellow}`, and another constraint says `{yellow, green}`, then the merge method gives a combined patch that must be in `{yellow}`. Another (trivial) example is the merge method for a scenario where we know that only one constraint will be affecting a given variable. The same situation holds for local propagation solvers: since a single constraint determines the value of the given variable, this merge method simply takes that patch and returns it. The smarts in this case comes from the satisfaction algorithm, which decides which variable to compute with which constraint, and in which order to do these computations.

5.4 Level 3: Satisfaction Algorithm

The job of the satisfaction algorithm is to decide on an order for updating the constrained variables, and for each variable, which constraints to consider in generating the merged patch.

The generalization of van Overveld's algorithm is one example of a satisfaction algorithm. Here, we consider all the constraints in parallel, feed the patches to the merger, which takes their weighted average, and then simultaneously updates all of the variables. The process continues until it reaches a fixed point, which will be a local minimum. If the constraints are required and there is still a residual error above ϵ, the system should raise an exception.

Local propagation solvers, for example DeltaBlue [6], also fit into this framework. For DeltaBlue, the satisfaction algorithm is in charge of figuring out the correct order to consider the constraints and variables. Given this ordering, for each constraint there will be a distinguished variable whose value will be found by that constraint. We then get a patch just for that variable and apply it.

We can also implement a simple finite domain solver using the framework. Here, the variable values are *sets* of possible values for the variable, from a finite universe. The merge method here is as described above. The algorithm iterates around until we reach a fixed point (i.e., no more restrictions) or until we have unsatisfiable constraints. At this point, if all the variables don't have a single value, we need to try *labeling* them, i.e., trying a specific value and seeing if it is possible to satisfy the constraints on several variables.

6. Execution Model

In the previous section, we provided the definitions for a general constraint solving framework. However, SKETCHPAD14 currently implements a more restricted form of CDP. While the programmer is allowed to implement the first (individual constraints) and second (merging solutions) levels, the top level (satisfaction algorithm) is currently fixed to perform the relaxation method. Customization of the top level is left as future work. In this section we delve into how the current system works internally.

There are three main stores that are maintained by the model: (a) **data**, (b) **constraint**, and (c) **event** stores. These stores get populated and modified by user operations of instantiating or removing data and constraints, or registering events handlers. During the execution of the program, the stores might be modified as part of solving for various constraints or running event handlers.

In order to support responsive interactive and animated applications, we adopt a two-phased execution model (depicted in Fig. 8), cycling at an adjustable frame rate. The clock time at the moment of each arrival on the first phase is used as the *pseudo-time* for the system. The two phases are the **event handling** phase and the **solving** phase.

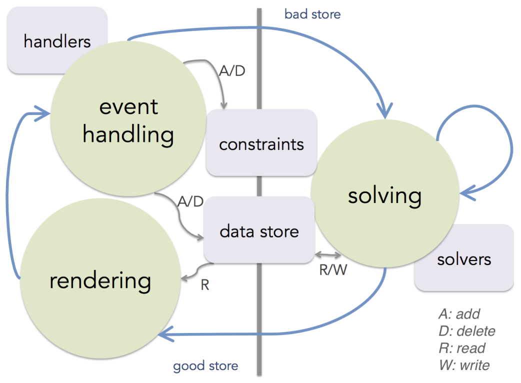


Figure 8. Two-phase execution model in CDP

6.1 Event Handling Phase (A)

The system starts in the event handling phase, at top of which it is considered to be at a "good" state (constraints are satisfied when possible), and the current state is drawn on the canvas (should changes have occurred). The **rendering** step calls the `draw` methods for all objects to refresh the frame. At this time the system updates its pseudo-time to reflect the time passed since last cycle. The handlers for any cached events that may have occurred during the previous solving phase are executed next. During such step the data and constraint stores may be modified. As a result (or despite of the fact) some constraints in the constraint store may now be unsatisfied. In any case, once all relevant handlers are executed, the system moves onto the phase *B*.

6.2 Solving Phase (B)

During the solving phase (illustrated in Fig. 9) the system invokes the `error` function belonging to each of the existing constraints. As long as the value is more than an ϵ error threshold, the system invokes the `solve` function associated with each unsatisfied constraint. Note that because at this stage solutions are simply collected but none is actually applied, all constraints will see the same state.

At this point the system uses the designated *merge* functions to combine all solutions to each variable into one consolidated new value, and then applies it to the variable. However, since these values won't necessarily be what each constraint had asked for, after this step all or some of the previously unsatisfied constraints are likely to remain unsatisfied (or even previously satisfied constraints may have now turned unsatisfied). As long as that is the case, the system applies iterative relaxation, repeating the above process, until either: (1) no unsatisfied constraints (computed error $> \epsilon$) remain, (2) the sum total error from all constraints remains unchanged for several consecutive iterations (reached a *fixed-point*), or (3) the solving phase times out, that is, the system has remained in the solving phase for longer than the frame rate value. At that point, the model cycles back to the event handling phase, and this process repeats.

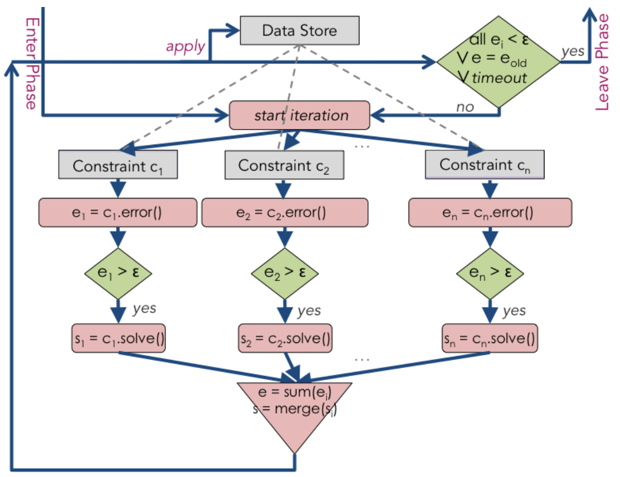


Figure 9. Solving phase

It is important to note that, because the solving is in the hands of the programmer, the process isn't guaranteed to succeed. It may get stuck or diverge. In the previous section we defined the conditions that ensure the process successfully converges to a solution.

7. Developer Side: Creating Applications from Scratch in SKETCHPAD14

In Sec. 4 we saw how existing building blocks can be used to construct a program and in Sec. 6 we learned about the system under the hood which executes it. Let's now discuss building new blocks, i.e., data or constraint types, which are often necessary when making a new application.

7.1 Parable of the Polygons

Recently, CDG's Vi Hart and web programmer Nicky Case released a highly cited and praised web page called "Parable of the Polygons."[4] Not only beautifully designed, the page explains a socioeconomic concept in an interactive and explorable way, and leads to conveying a positive social lesson of tolerance. We took on the task of remaking the web page in SKETCHPAD14 and the CDP model to learn their advantages and limitations. This section summarizes making this demo, a snapshot of which appears in Fig. 10.

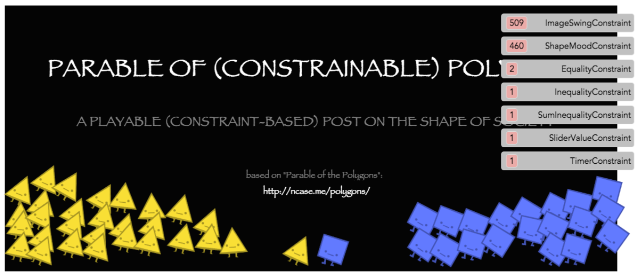

PARABLE OF (CONSTRAINABLE) POL
A PLAYABLE (CONSTRAINT-BASED) POST ON THE SHAPE OF S
based on "Parable of the Polygons":
http://ncase.me/polygons/
509 ImageSwingConstraint
460 ShapeMoodConstraint
2 EqualityConstraint
1 InequalityConstraint
1 SumInequalityConstraint
1 SliderValueConstraint
1 TimerConstraint

Figure 10. *Parable of the Polygons* remake

Though this demo does not fully implement the original page, most interesting and interactive behaviors are present. Surprisingly, only three new constraint types had to be defined, describing behaviors very specific to this page. The other six types were more general and already part of our predefined set of constraints or present in previous demos.

All CDP programs are conventionally organized in the same way. **Data classes** are defined first, then **constraint classes**. Afterwards **data** objects are created and then **constraints** are instantiated to operate on them. Finally, should this be an interactive program (all SKETCHPAD14 programs are because objects are draggable), **events** are registered, whose job is to modify the data and constraint stores.

7.2 Data Classes

In the page there are many polygon shaped objects (triangles and squares) which are draggable. We defined a `Shape` class to denote each shape in the page. Each one can be part of a `Board`, another class defined. A `Board`'s `Shape` has a `mood` (*sad*, *meh*, and *yay*, based on how many of its neighbors are of the same kind) determining its rendered `image`. A `Board` has a `cells` property, an array representing what's on it, whose elements may be `Shape`s or none.

We will see later than one of the constraint types will be affecting the `mood` property of a `shape` and another the `url` property (a text) of its `image` property. As we discussed, the system needs to be told how to combine multiple solutions from all constraints affecting each variable. The merge function can be associated with a class property. For both these non-real valued properties we designate the trivial merge function that expects only one solution. One of the constraint types will also be affecting the `cells` array property of a `Board`. For this, the class uses a different merge function called `dictionaryAddMergeFn`:

```
dictionaryAddMergeFn = function(curr, sols) {
  sols.forEach(function(dict) {
```

[4] `http://ncase.me/polygons/`

```
    for (var k in dict) curr[k] = dict[k] })
  return curr }
```

where each solution is a dictionary of key-value pairs that includes keys (in this case indices in an array) that it wants to change. Now the class defines its `merge` function:

```
Board.prototype.merge = function() {
  return {cells: dictionaryAddMergeFn} }
```

The above works since we know there will be exactly one constraint that will want to set the `cells` property of a board at a given time, so there'll be no conflicts.

7.3 Constraint Classes

We define a few constraint types each enforcing a certain behavior needed in the page. We'll preview one here and leave the rest for the appendix Sec. A.2.

`ShapeMoodiness (Shape shape)`: states that a `shape`'s `mood` property must be set based on the surrounding shapes (see the original webpage). Here are the definitions related to property type checking and description generation:

```
ShapeMoodiness.prototype.propertyTypes =
  {shape: 'Shape'}

ShapeMoodiness.prototype.description =
  function() {
    return this.shape + "'s mood is set based"
    + " on its neighbors." }
```

And now things related to constraint solving:

```
ShapeMoodiness.prototype.predicate =
  function(pseudoTime) {
    return this.shape.mood ===
      this.shape.board.getMood(this.shape) }

ShapeMoodiness.prototype.solve =
  function(pseudoTime) {
    return {shape: {mood:
      this.shape.board.getMood(this.shape)}} }
```

`ShapePlacement (Shape shape)`: states if `shape` is moved inside its board and fits in the dropped coordinate, it should snap in place in the center and its board should add it in the right index of its `cells` array. Otherwise it should snap in its original position. Appendix Sec. A.2 covers the implementation of this constraint.

7.4 Reactive Parts

We'll discuss one example of describing interactive/reactive behaviors here and leave the rest for the appendix Sec. A.2.

Placing pieces on a board (on `mouse` events): Since `ShapePlacement` constraint applies only when a piece that was picked up is dropped by the user, it makes sense to only add this behavior on the fly and remove it the next time another piece is picked up (since by that time that original piece should already be snapped in the right place):

```
registerEvent('mouseup', function(e) {
  var thing = e.pointedObject
  if (thing instanceof Shape
    && thing.board !== undefined)
      placementConstraint = addConstraint(
        new ShapePlacement(thing))})
registerEvent('mousedown', function(e) {
  if (placementConstraint !== undefined) {
    removeConstraint(placementConstraint)
    placementConstraint = undefined
  }})
```

7.5 Final Step: Laying Out Data and Constraints

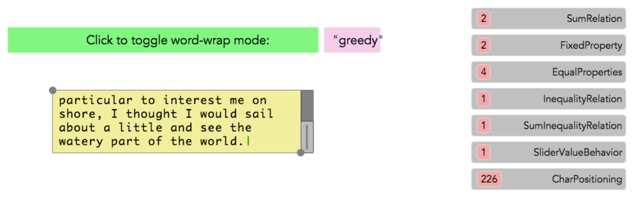
Once all kinds of data and continuous and reactive behaviors have been defined, we finish up by laying out the program: instantiate as many `Shape`s, `Board`s, `TextBox`es, etc. as we need, and add constraints as appropriate; e.g., all shapes on a board get a `ShapeMoodiness` constraint, and so on.

8. Experience

SKETCHPAD14 is open source, available online, and playable in the browser at `http://www.cdglabs.org/sketchpad14/`. There you'll find also a blog post as an extended version of this paper, with interactive graphics and embedded demos.

We have made numerous demos all fully implemented in the CDP model—that is, no fallback to JS code at the top level, and all JS code at the implementation level of constraints. These include a toy version Text editor with various word-wrap modes (Fig. 11), remake of the "Parable of the Polygons" web page (see Sec. 7 and 8), a Sudoku application with solving features (Fig. 12), a word game called "Quick Brown Fox", a puzzle game of Pentominoes with solving capabilities, and physics simulations (e.g. see Fig. 16). With minimal effort we also added a 3D frontend to SKETCHPAD14, using the nicely packaged *Three.js*[5] library (see Fig. 13). Finally, we successfully implemented in this framework a selection from the 7GUIs benchmark[6] (Fig 14), a popular GUI/web programming benchmark. All demos can be found in the tool. Please click on the *Example Bin* button to see the list.

Our overall experience in making new applications has been quite positive. The familiar obstacles and impracticality when performing all or part of the computation using constraint solving did not come up.



```
particular to interest me on
shore, I thought I would sail
about a little and see the
watery part of the world.
```

2	SumRelation
2	FixedProperty
4	EqualProperties
1	InequalityRelation
1	SumInequalityRelation
1	SliderValueBehavior
226	CharPositioning

Figure 11. Text area with "word-wrap" modes, e.g., one employing dynamic programming for optimal line breaking

[5] `http://threejs.org/`

[6] `https://github.com/eugenkiss/7guis/wiki`

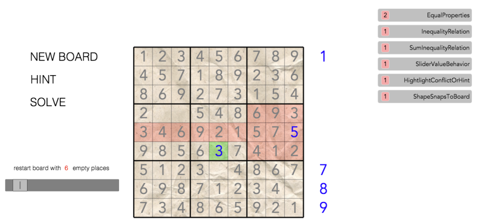


Figure 12. Sudoku

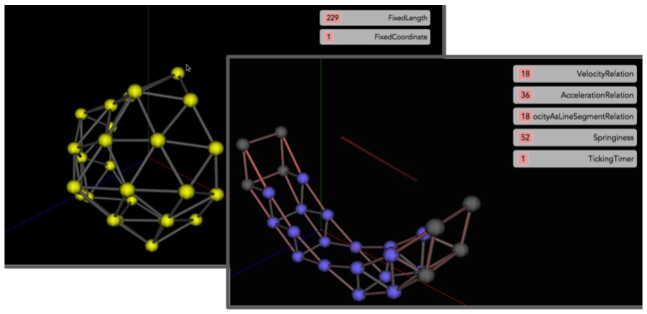


Figure 13. SKETCHPAD14 3D: Geodesic dome & bridge

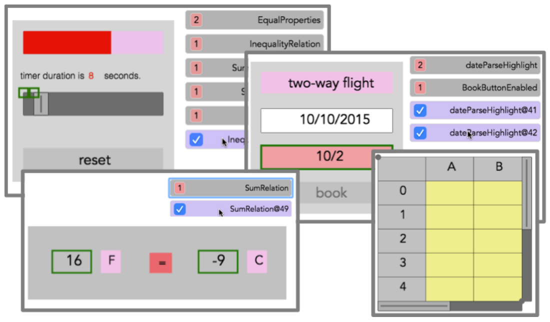


Figure 14. Snapshots from the 7GUIs benchmark

To learn more about the merits of our approach, we compared it with both imperative style and existing constraint-based programming, in the context of several of our benchmark applications, including the *parable* discussed in Sec. 7.

8.1 Advantages

In CDP behaviors stand out; in IP they get buried: Imperative programming (IP) doesn't enforce any particular restraint on programmer's thought process or program organization. He is free to write any code as long as it eventually captures the desired behaviors. Yet the resulting code, although functional, can be cryptic or mix behaviors in a way that's hard to parse to human mind. Moreover, there is no easy way to explore and understand the program by shallow inspection due to the lack of any high-level, semantic organization.

Take the slider in the *parable* page, for example, which is used to set the *bias level* of a shape. Where is the piece of code that's responsible for making such functionality? After digging into this, we see that the part of the page that includes the slider is actually embedded as an *iframe*, which loads another HTML file. There we see that there is an event listener, with a callback setting the associated global variable representing bias to the value of the slider:

```
onChange: function(values) {
  window.BIAS = values[0];
  // some more code...
  bias_text.innerHTML =
    Math.round(window.BIAS*100)+"%"; }
```

The name of the global suggests we have identified the code responsible for this behavior. But there was no indication anywhere. After all, these are just lines of code like many other thousands of lines of code in this application.

The CDP model, however, forced the programmer to express this behavior as a constraint type. The slider behavior indeed clearly shows up in the list of constraints as an enabled behavior in the scene. Here we clearly see the `SliderValueConstraint` in the list, can hover over the instance to see the objects it's applied to (Fig. 15), and can temporarily disable the constraint to verify the behavior.

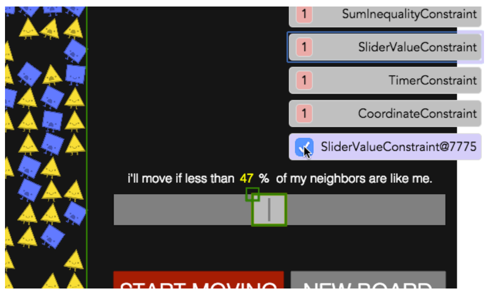

1 SumInequalityConstraint
1 SliderValueConstraint
1 TimerConstraint
1 CoordinateConstraint
SliderValueConstraint@7775
I'll move if less than 47 % of my neighbors are like me.
START MOVING
NEW BOARD

Figure 15. Exploring the slider behavior

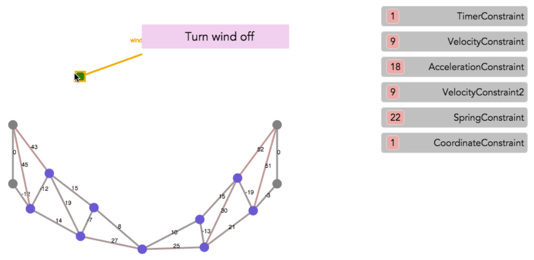
Also, despite being an elaborate demo, the set of behavior types are relatively short (8-9) as we see in the graphic. Thus the automatically generated English description (see Sec. 4) summarizing the active constraints gives a digestible overview of the full program. This can be contrasted to many thousands of lines of code which make the original JS implementation of the *parable* page.

CDP induces a "linear" thinking process in design & implementation; IP results in "ad hoc" ways: The CDP model forces the programmer to think and design the program as a set of individual and orthogonal behaviors. Once the set of behaviors are defined (as constraint types), building a program is simply a matter applying a selection of them to the data. In our experience, this particular program organization actually guided and simplified our thinking and design process, easing the overwhelming effects of making applications from scratch. Similarly, the non-linear interactions that are part of interactive applications are tamed by the model. It provides a principled manner to register events and handlers, and makes the actions that need to be done by the handlers straightforward, which is often simply adding or removing constraints.

By contrast, in IP the process is quite ad hoc. In the lack of any organization or guidelines, the code often makes no attempt to keep the behaviors separable and modularized. This can also confuse the thinking process itself, leading to bugs. Finally in IP extensions and modifications may require algorithms to be reworked.

CDP reduces the burden on the programmer having to reason globally and computing or describing "emerging behaviors": Have a look at the truss bridge demo shown in Fig. 16, an original Sketchpad demo and remade here using SKETCHPAD14. Although it cannot be seen with this static image, the resulting emergent behavior of the program simulating an oscillating bridge is quite elaborate. There is not an easy way to "code" the behavior of this bridge under the forces of gravity and wind imperatively. On the other hand, as you can see on the constraint listing on the right pane, the constraint types that make up this overall behavior are not complicated at all. We only had to define behaviors of velocity, acceleration, and the spring force (Hook's law).

The particular program structure in CDP enables the programmer to focus more on "local" reasoning (individual orthogonal behaviors) and worry separately on the "global" logic of how local solutions should interact. We simply had to instantiate these constraints for all the bodies and beams (springs) that make up the bridge. Because the particular constraints involved here were on real-valued variables, a simple van Overveld-relaxation-style *averaging and iterating* was enough to let the system capture the global emergent behavior by combining the results of individual local constraints. In our experience, even when relaxation-style solving was not appropriate, the simple fact that the local and global reasoning are separated helped simplifying and reasoning about the design.



1	TimerConstraint
9	VelocityConstraint
18	AccelerationConstraint
9	VelocityConstraint2
22	SpringConstraint
1	CoordinateConstraint

Figure 16. Emergent behavior: oscillating bridge under natural forces. See it in action at `https://vimeo.com/107522156`.

CDP enables "modular" behaviors: Next, take a look at Fig. 17 (left). Within the *parable* demo we have loaded another demo, called *mid point*, where a red dot always is maintained at the middle between two blue dots. Fig. 17 (right) shows the effect after we have *merged* these three dots' behavior into three of the shapes from the *parable* example. The shapes retain their previous qualities and behavior (swinging, etc.) yet also take on the new behavior of the *mid point* example, with the blue square always staying at middle point between the two yellow triangles.

In terms of CDP this operation is quite simple. All constraints previously operating on the three dots (easily enumerable by inspecting the constraint store) were respectively transferred to the three shapes. Then old dots were deleted. Doing this in IP is by no means easy. Where to add this new behavior? How to make sure previous behaviors still work?

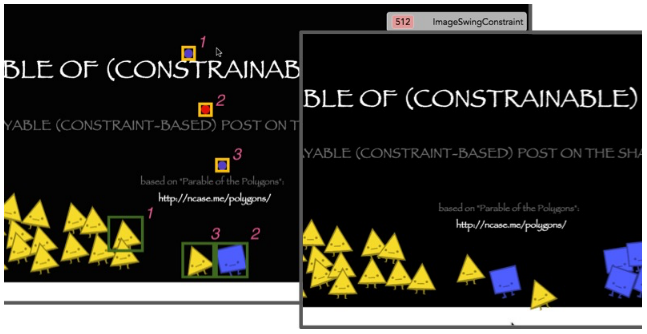

512 ImageSwingConstraint
BLE OF (CONSTRAINAB
YABLE (CONSTRAINT-BASED) POST ON T
based on "Parable of the Polygons":
http://ncase.me/polygons/
BLE OF (CONSTRAINABLE)
YABLE (CONSTRAINT-BASED) POST ON THE SHA

Figure 17. Modular behaviors using merge: (*left*) before (*right*) after

CDP makes IP compatible with constraint programming: IP and declarative programming do not mix well. Hybrid systems such as constraint imperative programming [5, 7] have been proposed, though the constraints there have a specific role and do not replace the imperative code.

The CDP discipline forces imperative programs to be organized in a way that at the top level a program is simply composed of a set of constraints. This setup enables a seamless usage of traditional constraint-based programming as part of the implementation of a program. The user can easily decide to invoke an external constraint solver as part of the `solve` function for a specific constraint, while other constraint types may still be solved using imperative code, or even perform global constraint solving using an external solver. For example, in our *text editor* application, we used a well known dynamic programming algorithm as the solution for a constraint that requires the line-breaks in a paragraph be chosen in a way so that the variations in margin size of each line are minimal. We believe developers will find this level of flexibility quite compelling.

Truly unified constraint & programming language: One of the barriers in traditional constraint-based programming is the lack of unification between the language of the constraints and that of the rest of the program. The mismatch is plain to see if we attempted to solve the constraints involved in the *parable* application in Prolog, for example. It is unclear if it is even possible, and even so taking on the task of translating of the intentions into constructs of Prolog seems daunting. The mismatch is less felt in a constraint-imperative language, such as Babelsberg [5, 9], since a subset of the programming language is itself used to construct constraint expressions, which the system automatically compiles to the encoding of the supported external solver. Nevertheless limitations and mismatches still exist, since we are necessarily bounded by the expressive limits of the solver.

A nice property of CDP is that the entire program, including the declaration and solving of the constraints, is done in one language. As a result, the user can use the same data structures, abstractions, methods, etc. to express constraints and provide the logic and strategies to solve them.

9. Limitations

We face important limitations in our proposed programming model in its current form, discussed below.

Lost guarantees of constraint solving: An essential appeal of constraint-based programming is that when the solver does return a solution, all considered constraints are guaranteed to have been satisfied. The relaxation method of Sketchpad and van Overveld are weaker; convergence is not guaranteed and the system may get stuck in a local minimum. However, it is guaranteed that with each iteration the total error will not increase.

The biggest flaw of CDP is that the guarantees of global constraint solving (as well as the property of relaxation method stated above) are lost, as soon as we have opened up to the programmer both the algorithm to solve each constraint as well as the logic of how the set of solutions are combined. The system currently is able to signal to the user if upon the completion of the solving phase any constraints remain unsatisfied. However, no automatic remedy is available. It is up to the programmer to verify that the overall outcome meets the expectation.

Our Initial focus so far has been on allowing the generality and flexibility to successfully implement our benchmark applications. Nonetheless, as part of future work, we would like to be able to provide stronger solving guarantees under restricted scenarios.

Lack of flexibility over the overall solving process: As we have seen earlier, when dealing with finite-domain or integer constraints the van Overveld approach of averaging solutions to consolidate does not work. We thus let the user provide the logic of how to combine solutions. Yet at the top level there is no flexibility; the system performs the iterative relaxation process. This process seems to be sensible in some but not all scenarios. Sometimes a finer level of control for the entire solving process is needed.

For instance, sometimes performing traditional backtracking search might be suitable (a proposed solution for that is found in appendix Sec. A.1). Sketchpad and ThingLab, besides the general relaxation scheme, employ a straightforward local propagation approach when possible. The Nelson-Oppen method for cooperating decision procedures [14], around which SMT solvers are built, has its own iterative approach to solve a problem involving multiple theories. As part of future work, we need to provide a framework where the programmer can control the overall solving procedure in a finer way, so that scenarios above or others could be accommodated without much difficulty.

Choosing the right granularity level for constraints: The choice of how large or fine-grained constraints are organized is important. Consider the example of solving a system of arithmetic linear inequalities. Clearly, setting up a separate constraint for each equation and then relying on iterative relaxation to converge the solutions from each equation is not the way to go! In that case, we have efficient solving tools, e.g. the *simplex* method, to simultaneously solve all the inequalities at once. It is nice that the CDP model gives us this flexibility; we simply create one constraint type that represents the system of inequalities, whose `solve` function will invoke a simplex solver to compute the answer. We have used the following rule of thumb when designing the constraint types to be used in a program:

> *Break a program into a set of logically orthogonal, or only loosely related, behaviors.*

The objective here is to reason about solving each constraint separately, and then in the `merge` function reason about how conflicting solutions should be consolidated. Nevertheless, more experience is needed to learn about the challenges in defining the right abstraction level for constraints.

Developer burdens: CDP enforces a particular way of organizing a program, a source of burden to developers. Moreover, unlike traditional constraint-based systems which make use of external solvers, the burden of solving the constraints also falls on the shoulders of the programmer.

Performance hits: Its execution model follows an involved process and all the machinery can be a huge performance overhead over a hand-coded imperative program that performs the precise set of necessary instructions to accomplish a certain task. For animations, the overhead of computation is exacerbated by the fact that we might repeat the process at a rate of 60 times a second.

Nonetheless, in our current set of animated demos the performance is satisfactory, i.e., smooth (see the truss bridge demo referred to earlier). We also have leveraged easy optimization opportunities in the tool. For example, by simply keeping track of the total error of the system we detect convergence to avoid redundant computations.

10. Related Work

This section presents a brief summary of related literature on constraint-based programming.

10.1 Pure Constraint Systems

Sketchpad [16] and van Overveld's relaxation approach [18] were our main inspiration for the CDP model and SKETCHPAD14, by looking beyond the focus there—geometric constraints. ThingLab [3] made an important step in building on its ideas within a general object-oriented language.

Pure constraint languages have failed to pervade mainstream software development, lacking severely when it comes to scalability and practicality. The aim of many, including us, has been to leverage the full expressiveness and power in imperative languages while bringing in some of the benefits of declarative and constraint-based approaches. Some have also applied the ideas on a specific aspect of development, such as layout in graphical applications.

10.2 Mixed Systems

Let us now focus on hybrid programming models.

10.2.1 Declarative within Imperative

Languages that bring some form of constraint-based programming within an imperative system have a long history; see for example constraint-imperative programming (CIP) [5, 7, 12] or related work [1]. The convention is that while we're still working in an imperative (e.g., object-oriented) language, a constraint language and solver is used to let the system automatically maintain a set of relationships. For example declaring `y = x + 1` should ensure that, on subsequent assignments to either variable due to the execution of some statement, the system will automatically update the other accordingly.

The hybrid form of declarative within imperative style programming we propose here, however, is totally different because the two apply at different levels. At the highest level the program is only declarative, consisting of a set of constraints. Imperative code, however, is used at the lower, implementation level: for the purposes of solving of individual constraints and combining their results.

The problem with the conventional mixed paradigms [5, 7, 9, 12] is that many of nice properties of pure declarative programming are lost. While constraint solving does automate and relieve the burden of maintaining some of the dependencies upon updates, a program cannot be fully explained or understood just in terms of constraints. Both the imperative code and the declarative constraints need to be inspected to account for the full behavior. The existence of imperative code means the existence of its flaws in modularity, extensibility, understandability, and debuggability.

The major downside of CDP, as we mentioned, is that the outcome of constraint solving cannot be guaranteed. This is because individual constraints are solved and combined as programmed by the developer. In CIP, on the other hand, the declared constraints are guaranteed to hold, whenever the solver is able to find a solution.

In CIP and other forms of mixed constraint programming, the constraints are only expressed, and it is the job of the system to solve them. This can be a good or a bad thing! It's good if the user can get the desired behavior; but it's bad if the system is unable to support solving the constraint (at all or efficiently) and provides no mechanism to help or steer the process. In CDP, however, we trade automatic solving for control, by letting the programmer, separately from the expression of the constraints, provide the logic for solving them, by writing arbitrary code.

In some forms of CIP, e.g. the Babelsberg family of *object-constraint* languages [5, 9], constraints are also expressed using the same host language rather than a DSL. However, constraints can only be expressed in terms of low-level operations which the *constraint construction mode*, a form of hybrid concrete and symbolic execution, and the underlying solver can support. For example, in Babelsberg/S [9] basic `Array` class's low-level access methods had to be redefined to support encoding constraints on collections. The language of the employed solvers is often limited in the set of supported data types. In CDP, however, there is no translation to an external solver entity, and thus we incur no restrictions on the set of data types/structures supported, as the solving logic is given in the host language.

Rosette [17] is a DSL design framework in Racket that specifically addresses the translation difficulties that arise in embedding automatic "solver-aided" features (e.g., verification, angelic non-deterministic execution) in designing a language. The goal of our work is quite different. We aim to provide a programming mechanism to build an entire application declaratively.

10.2.2 Imperative within Declarative

The reverse direction of mixed paradigms, that is, imperative within declarative languages, exists but is less common. Constraint systems that support *local propagation* (e.g., DeltaBlue [15]) can fall into this category. Given a multi-way constraint, these systems build a propagation graph that given the context tells which direction the data flows and how to compute the unknowns from known data. Some such systems allow the programmer to code each propagation scenario for each constraint. This is similar to the proposed model here where the developer implements the solving of a constraint using imperative code.

Modelica [8] is an object-oriented declarative modeling language, where equations are used to declare relationships in the model. Numerical methods are used for solving. The language also supports an "algorithm" construct to perform a computation imperatively when simpler. The language is primarily used for modeling, verification, and simulation of physical systems, however, not for developing software.

10.3 Domain Specific Systems

Some prior works concern declarative web programming, focusing on some aspects of web development, including forms [19], data [10], sessions, authentications, security [11], etc. Building interactive applications declaratively was an aspect of Elliott's functional reactive programming (FRP) [4], with followup works focused on web programming [13]. Numerous domain-specific declarative systems exist. Our aim, however, was to devise a general programming paradigm that would cover all aspects of interactive, graphical, and web development, so that the entirety of an application, from a top view, is described fully declaratively.

As a final remark, we have not seen any prior work used to make realistic, fully-declaratively designed applications such as those we have been able to write (e.g., a real-world high quality interactive web page) in SKETCHPAD14.

11. Conclusion

We presented a programming model as a pragmatic approach to constraint-based programming which aims for the practicality and scalability of imperative programming, while still leveraging some of the great benefits of declarative constraint-based programming such as raised level of abstraction and organization, understandability, extensibility, and modularity.

In this model programs are organized in a way that at the top level only constraints exist. Yet the solving of constraints is orchestrated by the programmer and within the imperative language. We empirically illustrate the gained benefits in comparison to traditional imperative and constraint-based programming through several realistic applications. Our hope for this work is that it is not only an academic interest, but one that serves software developers by making declarative programming more accessible.

Acknowledgments

We thank Alan Kay for his lead to revisit Sketchpad and Todd Millstein for helping shape the narrative of this paper.

References

[1] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. Technical report, Amsterdam, Netherlands, 1997.

[2] A. Borning and B. Freeman-Benson. Constraint Hierarchies. *LISP and Symbolic Computation*, 5(3):223–270, 1992.

[3] A. H. Borning. *Thinglab–a Constraint-oriented Simulation Laboratory.* PhD thesis, 1979.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP '97.*

[5] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/js: A browser-based implementation of an object constraint language. In *ECOOP 2014 Object-Oriented Programming*.

[6] B. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, Jan. 1990.

[7] B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *OOPSLA '90.*

[8] P. Fritzson and V. Engelson. Modelica a unified object-oriented language for system modeling and simulation. In *ECOOP 98.*

[9] M. Graber, T. Felgentreff, and R. Hirschfeld. Solving interactive logic puzzles with object-constraints. In *Workshop on Reactive and Event-based Languages and Systems*, 2014.

[10] M. Hanus and S. Koschnicke. An er-based framework for declarative web programming. In *PADL'10.*

[11] T. L. Hinrichs, D. Rossetti, G. Petronella, V. N. Venkatakrishnan, A. P. Sistla, and L. D. Zuck. Weblog: A declarative language for secure web development. In *PLAS '13.*

[12] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL '12.*

[13] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA '09.*

[14] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2): 245–257, Oct. 1979. ISSN 0164-0925.

[15] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Softw. Pract. Exper.*, 23(5):529–566, May 1993.

[16] I. E. Sutherland. Sketchpad a man-machine graphical communication system. In *Papers on Twenty-five Years of Electronic Design Automation*, 25 years of DAC.

[17] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Onward! 2013.*

[18] C. van Overveld. 30 years after Sketchpad: Relaxation of geometric constraints revisited. *CWI Quarterly*, 6(4):363–383, 1993.

[19] W3C. Xforms 2.0. `http://www.w3.org/TR/2012/WD-xforms20-20120807/`.

A. Appendix

An addendum to the programming model proposed in the paper as well as more details on the implementation of one of our benchmarks are presented here.

A.1 Enabling Backtracking Search

Classical backtracking search isn't accommodated by the CDP model as described. However, we have made a modification that allows programmers to rely on search, when suitable, as part of a program.

We let a constraint type to be marked as `searchable`. A searchable constraint's `solve` function returns a set of solutions rather than one. Thus a user can communicate to the system that it wants to try either one of the solutions and search for the right combination of solutions from all constraints that will result in a globally acceptable solution.

During the first iteration of the solving phase *B*, the system collects the set of solutions from each searchable constraint (typically representing all permutations to try), as well as the usual one solution from others. It then constructs a search tree that gives all possible combinations of choosing a single solution from each constraint. (When no searchable constraint exists this tree has a single branch as the only combination to explore.) At this point the system engages in a brute-force search to find the combination that leads to a satisfiable place. For each combination choice, it does the usual relaxing-style iterative solving up to some timeout duration, except that searchable constraints do not participate during those iterations. Should a fixed-point be reached with all constraints satisfied, the solution is committed and the model cycles back to the reactive phase *A*. Otherwise, solutions in the last attempt are discarded and the search of all possible combinations continues on.

As an example, below we have defined a constraint called `NumListSorted` to ensure a list of numbers is sorted. Provided that a copy of the list is kept in `oldList`, the `predicate` function returns `true` only when the list has been properly sorted. The `solve` function simply returns all possible permutations of the list, therefore hinting to the system to search for one that would make the constraint satisfied.

```
NumListSorted.prototype.searchable = true

NumListSorted.prototype.predicate =
  function(pseudoTime) {
    return sorted(this.list)
      && isPermutationOf(this.list, oldList) }

NumListSorted.prototype.solve =
  function(pseudoTime) {
    return allPermutations(oldList).map(
      function (l) { return {list: l} })  }
```

Note that the above scheme works well when the numeric constraints solved by relaxation and finite domain constraints solved by search are mixed together within the same program, provided that numeric constraints use the searched-over values in a read-only manner. Roughly speaking, if the relaxation-style iterative approach is like *hill-climbing* in some neighborhood in the state space, our augmented search-model is akin to simultaneously running hill-climbing in different regions in space in order to have a better chance of finding a solution.

A.2 Parable of the Polygons: More Details

Some of the remaining continuous and reactive constraint definitions from the *parable* application are listed here.

A.2.1 Constraint Classes

`ShapeSwinging (Shape shape, Number swingSpeed, Boolean dangling)`: states that `shape` must swing back and forth over a span of 72 degrees, at a rate of `swingSpeed`. We will see later how `dangling` boolean flag is used. `this.image` will be set as `shape.image`.

These are the declarations related to constraint solving:

```
ShapeSwinging.prototype.error =
  function(pseudoTime) {
    this.targetRotation =
      this.image.origRotation
        + (Math.sin(this.swingSpeed
          * pseudoTime) * Math.PI / 10)
    return this.targetRotation
      - this.image.rotation }

ShapeSwinging.prototype.solve =
  function(pseudoTime) {
    return {image:
      {rotation: this.targetRotation}} }
```

`ShapePlacement (Shape shape)`: states if `shape` is moved inside its board and it fits in the dropped coordinate, it should snap in place in the center and its board should add it in the right index of its `cells` array. Otherwise it should snap back in its original position.

`error`: We check if the position of shape is within the board and if the board coordinate closest to the shape's position is free. If so, we compute the position where the shape should be snapped onto. Otherwise, it should snap back at its original place on the board (`this.shapeOrigPos`). The error value is the distance between the current position of the shape and its computed target position:

```
ShapePlacement.prototype.error =
  function(pseudoTime) {
    var currPos = this.shapePos
    var board = this.shape.board
    this.shapeCoord = board.getCoord(currPos)
    var inside = board.containsPoint(currPos)
    if (inside && board.fits(shapeCoord)) {
      this.placing = true
      this.targetPos = plus(board.position,
        {x: shapeCoord.j * board.cellLength,
         y: shapeCoord.i * board.cellLength})
    } else {
      this.placing = false
      this.targetPos = this.shapeOrigPos
    }
    return magnitude(
      minus(this.targetPos, currPos)) }
```

solve: We have already computed where the position of shape must be updated to and stored it in this.targetPos. If the shape is being placed on the board (captured by boolean value this.placing) then the solution wants to free up the previous cell location and set the content of the new cell coordinates to this shape. This is done by setting the respective location in the cells array to 0 (for *empty* cell) or the shape, respectively:

```
ShapePlacement.prototype.solve =
  function(pseudoTime) {
    var board = this.board, shape = this.shape
    var sol = {shapePos: this.targetPos}
    if (this.placing) {
      var shapeOldCoord = shape.boardPos
      var dict = {}
      dict[(shapeOldCoord.i * board.width)
        + shapeOldCoord.j] = 0
      dict[(this.shapeCoord.i * board.width)
        + this.shapeCoord.j] = shape
      sol.board = {cells: dict}
    }
    return sol }
```

A.2.2 Reactive Parts

And now we complete the listing of reactive parts of the demo.

Dragging pieces (CoodinateConstraint on mouse events): As with all demos, the default dragging-related events allowing things to be moved are in place.

Swinging faster on mouse hover (ShapeSwinging on mouse events): The pieces on the header of the page (stored in swingingShapes array) swing faster when mouse hovers over them. They all already have a ShapeSwinging constraint operating over them, so it's simply a matter of changing the property of that constraint to reflect the speeding or slowing of the swing rate:

```
registerEvent('mousemove', function(e) {
  swingingShapes.forEach(function(shape) {
    var dist = distance(e.mousePosition,
      shape.position)
    // kept a reference to the constraint:
    shape.swingConstraint.swingSpeed =
      2 + (dist < 200 ?
        ((200-dist)/20) : 0) } })
```

Pieces dangle when dragged (also using ShapeSwinging on mouse events): We'll add an instance of ShapeSwinging to not only the happy swinging shapes on the page header, but also to a shape that is picked up by the user, for a dangling effect. The difference is that the dangling-kind swinging rate subsides with the passage of time.

We use a feature in SKETCHPAD14 not yet discussed. Each class (be it a data or constraint type) can define an onEachTimeStep method, which will be run by the CDP execution model right before going to the solving phase *B*, and is used to do any sort of housekeeping, property updating, etc. Thus we define one such method for each instance of the ShapeSwinging which happens to be simulating dangling, in order to get the effect of gradually ceasing the dangling.

```
ShapeSwinging.prototype.onEachTimeStep =
  function(pseudoTime) {
    var shape = this.shape
    if (this.dangling) {
      var movement = shape.position.x
        - shape.lastPosition.x
      if (Math.abs(movement) > 0)
        this.swingSpeed += (movement / 200)
      else {
        this.swingSpeed /= 1.05
        if (this.swingSpeed < 0.001)
          this.swingSpeed = 0
      }
      shape.lastPosition =
      shape.position.copy()
  } }
```

Alternatively we could have chosen to embed the behavior of subsiding the swinging rate as part of the definition and solution to the constraint. But here we took the easy route.

Now all we need to do to get the dangling effect is to add a ShapeSwinging when a shape is picked up (similarly, the constraint gets removed when a shape is dropped down):

```
registerEvent('mousedown', function(e) {
  var thing = e.pointedObject
  if (thing instanceof Shape) {
    danglingConstraint = addConstraint(
      new ShapeSwinging(thing, 2, true))
  } })
```