# State of the Research: Refiner

Calvin Loncaric  
July 22, 2011

## Overview

Currently, the entire refiner is a series of steps which are run one after another. Some of these steps (the UniqueNamer and the RecognizedMarker) are somewhat uninteresting and exist only to perform certain cleanup tasks that the program expects. This document describes the other refiner stages: my efforts toward writing a search refiner to improve recognition and grouping quality.

As of this writing, my work on the SearchRefiner is not quite complete and tests show little to no improvement over the default recognition process. A description of why this may be is provided in the “Limited Success” section below.

## Description of the Search Refiner

The search refiner (so named because it “searches” for the most correct interpretation of a drawn circuit) consists of three main kinds of objects:

ISketchModificationProducer  
A sketch modification producer does only one important thing:

* List<ISketchModification> SketchModifications(FeatureSketch)  
  Get a list of modifications that *could* be performed on the given sketch.

SketchModification  
A sketch modification describes some change that can be made to a sketch. It implements the following key methods:

* void perform()  
  Executes the modification.
* double benefit()  
  Obtains a measure of how “good” it would be to perform the modification.

SearchRefiner  
This main class uses a sketch modification producer to refine the sketch. It repeatedly obtains a list of all possible modifications and chooses one using a search method (e.g. hill climb search or simulated annealing). Currently only the hill climb search method is implemented, but implementing another should only be a few lines of code.

## Implementation Details

The primary workhorse of our search refiner is the CircuitSketchModificationProducer, which produces modifications for sketches in the circuit domain. Each modification has a benefit which is computed based on an energy function we are trying to maximize. The kinds of modifications it is currently capable of producing are:

* Connect shapes: connect a wire with a disconnected endpoint to a shape missing a connection. The benefit depends on how far the connection is, so distant connections are very unlikely.
* Re-recognize: assign a different type to a shape. The benefit depends on (a) how much the context of the shape and its neighbors would improve, and (b) how much the recognition confidence would suffer as a result.
* Stroke steal: each shape has a chance to steal a substroke from a neighboring shape. The benefit depends on how much recognition would improve, and how much the context of each shape would benefit as a result of the new recognition.

The benefits of modifications it produces are based on an energy function we are trying to minimize:

Where C(s) is the percent of the shape’s context that is properly matched and R(s) is the recognition confidence of the shape.

## Implementation Limitations

There is no simulated annealing or other sophisticated search algorithm. Implementing it may help the search refiner find better interpretations of the circuit; these tests have not been performed, however.

The refiner cannot differentiate shape-to-shape connections as “input” or “output.” This is a limitation of the original order of recognition, in which circuit parsing (where input and output are determined) comes last. I would like to move this particular part of circuit parsing up before the refiner so it is accessible as another piece of information.

The energy function does not look at orientation at all, even though the orientation of nearby wires actually gives a really good indication of what the orientation of a gate should be. This could let us cross-check the connector and the recognizer, or even hint to the recognizer what the orientation SHOULD be in order to get a more reliable result.

Some specific modifications which have NOT been implemented but SHOULD be are:

* Disconnect shapes: remove a connection between two shapes. Benefit would depend on how much the context of each shape would change.
* Stroke shed: split a substroke off a shape and make a new shape out of it. Benefit would depend on context changes and changes in the recognition confidence.

## Improvements over Past Work

There are a number of reasons why this architecture for the search refiner is superior to how it was organized in the past. It has a number of *very* nice features:

### Clear documentation of heuristics

The past refiner made a lot of decisions which were sometimes good ideas, but often weren’t and had no justification. The only heuristic at work in the new architecture is the energy function; change the energy function, and you change the search refiner’s priorities.

### Separates “finding possible actions” from “deciding which action to take”

Previously, the refiner would both seek actions and perform those actions. This mess made it difficult to find where problems were happening. By making sketch modifications atomic we can easily output the list of actions in the order they were taken, as well as the exact numbers for why they were picked.

### Order-agnostic

The old refiner would perform context refinement followed by stroke steal followed by stroke shed. Tests had to be performed to determine which order of these operations was best. Now, sketch modifications can be performed in any order, making it possible for the system to find other (possibly better) interpretations of the circuit.

### Self-correcting

Every action has a corresponding “undo” action that could be executed. So if the refiner makes a bad decision, it has the potential to fix itself later.

*Here’s a simple example of why our architecture is hard to work with: we want to move a substroke from shape A to shape B.*

1. *Remove substroke from shape A.*
2. *Each endpoint connection to A now points to B.*
3. *For each endpoint connection, connect shape B to the connected shape and vice-versa.*
4. *If shape A is now empty, remove it from the sketch*
5. *Add substroke to shape B.*

*In order to make this action undoable, you would have to record all the original connections for both shapes and all their connected shapes, all the original endpoint connections, whether shape A got removed from the sketch, etc. etc. It rapidly becomes a nightmare. Much of the code ignores these problems and expects automatic correction later.*

## Limited Success

In this section I will document why I think refinement has repeatedly met with difficulty or failure. Some of this refers specifically to my work, but most of it is applicable to any potential refiner implementation.

### Architectural Limitations

Due to the complexity of the data structures involved, it is *very* difficult to “test” a particular action and then undo it, which I think would be one of the most useful features we could offer. The UI, for instance, struggles with this a lot in order to enable “undo” functionality. It doesn’t always succeed. This limitation has largely been overcome by the new Sketch.Operations API that was introduced in commit #5978, but it is still a problem.

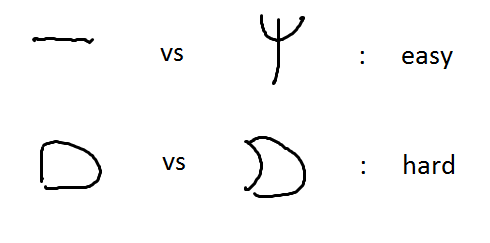
### Incomplete Implementation

The search refiner suffers as much from missing features as anything else. From what I’ve seen of previous work, similar problems were encountered. Writing a refiner like this is a large project, and one which I wasn’t able to even begin tackling until late in the summer. I was assisted greatly by the work of previous summers, but there was a lot missing that took too long to implement.

### Missing Confidence Measurements

Our two early stages (single-substroke classification and grouping) are unable to provide confidence measurements for their actions. When we get to the refiner stage, we can’t cross-check ourselves with those stages. The same goes for the connector. As the system stands now, we basically use the classifier to get to a starting interpretation, and then proceed to ignore it entirely as we search for a better interpretation.

### Limited Range of Confidence Measurements

General-purpose recognition algorithms like the image recognizer or a neural network are very good at seeing the differences between radically different shapes (like line and pitchfork on the right) and very bad at confidently differentiating similar shapes (like and-gate and or-gate on the right). As a result, the confidence measurements given to us by the recognizer tend to be very similar; it is only *slightly* more confident for some shapes. Because of this, it’s hard to tell whether a given recognition is wrong or not. While other shapes seem similarly likely to the recognizer, that information may be misleading. What we need is a recognizer that gives radically different confidence measurements for very similar shapes, but that is a very hard problem.

### C:\Users\research\Documents\Pictures\bad_recognition2.pngErroneous Confidence Measurements

Often the recognizer returns extraordinarily high confidence for things it has not seen before (like incorrectly grouped gates). An example I drew in the practice window is shown on the left. What’s happening here is very subtle, and is actually a limitation specific to the image recognizer. The image recognizer uses four coefficients to determine the best matching template. These four coefficients lie in different ranges, so in order to interpret them we normalize the values based on the values returned by the other templates. It helps us sort out “better” from “worse,” but doesn’t help us sort out “match” from “no match.” If a single template is the best match for all four coefficients—even if it is a terrible match—it will have a confidence of 100%.[[1]](#footnote-1)

### Heterogeneous Recognizers

One thing we would like to do is to ask questions like “is this shape more likely to be text, or a gate?” Unfortunately, this proves to be quite difficult. The text recognizer and the gate recognizer both return confidence measures, but they don’t necessarily mean the same thing. That is, the gate recognizer’s confidence is based on bitmap symbols, and the text recognizer’s is always 0.9, 0.5, or 0.1. These two confidence measures tell us different things, and can’t really be directly compared.

### Identical Shape Contexts

Out of the 7 major kinds of gates (and, or, nand, nor, xor, xnor, not), ONLY ONE has a different context than the rest. As a result of this and the limited range of confidence measurements (which I described above), we have almost no way to identify errors when recognizing and-gate versus or-gate. We have to depend entirely on the recognizer. This is reflected in my tests; with the search refiner we often misrecognize the other gates but identify not-gates correctly almost always.

### C:\Users\research\Documents\Pictures\2x valid contexts.pngSome Valid Contexts are better than others

I have not been able to come up with a good way of identifying which valid contexts are better than others. This is a broad problem, and only one small example is shown at right.

Consider the top xor-gate. We humans really want that blue wire on its back to be part of the gate itself. However, what the circuit parser sees is the second picture: that wire is connected to the gate at two points and it is (and should be) completely valid. The bottom picture is, in some ways, a “better” interpretation because it agrees more with the classifier. However, making that wire part of the xor-gate is intuitively a “better” interpretation because it “seems” more like what the user wanted. Given these two completely valid interpretations, though, how should the system know to pick one over the other?[[2]](#footnote-2)

## Results

These results were obtained on code revision 6061, by running the following command from /Trunk/Code/Util/TestRig/scripts:

..\bin\Debug\TestRig.exe -s p [p cls grp rec con "|" cls grp rec con ref\_search "|" cls grp rec con ref\_ctx] -n 50 -d "..\..\..\..\Data\DrawingStyleStudyData\All Complete Sketches" -contains .xml

The moral of the story is, I think, that the normal pipeline is still the best choice for now. Search refinement performs better than the old context refiner, but still isn’t ready for full use. More work is necessary.

## Conclusions

This is a good idea that should work. Various limitations hold us back. Here’s a list of what I think needs to happen before search refinement becomes possible:

* The classifier needs to emit a meaningful confidence measure for each substroke
* The grouper needs to emit a meaningful confidence measure for each pair of joined substrokes
* The recognizer needs to emit a meaningful confidence measure that uses the whole range from zero to one and is low for shapes it has not seen
* The connector needs to be able to differentiate between input and output connections

In the meantime we have shown that the following highly-specific refiner steps *do* improve recognition:

* Making missing connections even when they are outside the connector’s distance threshold
* Re-recognizing a shape as a not-gate if its context indicates it has only one input, or re-recognizing it as something else if it has more than one

Because of the “identical shape contexts” problem, I think perfect recognition will always be impossible. However, I see no reason why grouping couldn’t be dramatically improved by a refiner like this one, were it implemented correctly and if the various problems were addressed. I hope to continue working on this project a little during the Fall.

1. Here’s a fun project: keep using the normalization but add a neural network to sort out “match” vs. “no match” using the un-normalized coefficients. It’s the kind of thing a small neural net would be perfect for (approximating a simple function we don’t fully understand), *and* you could train the network on negative examples (i.e. “here’s what the coefficients look like for something that doesn’t match any template”). We have a neural network implementation already in the repository; this project would probably only take a few days to implement. Feeding a neural network highly processed values like this is a practice I’ve seen in some of the literature. [↑](#footnote-ref-1)
2. I think we may be able to solve some cases by valuing orientation of nearby wires in the energy function. After numerous user studies, we have not observed a single user draw wires coming into a gate at radically different angles. By favoring an interpretation where this is not the case, we could suggest to the system that the vertical wire is part of the xor gate. However, this doesn’t solve much if one of the *front* parts of the gate was misrecognized, so I have not taken the time to implement it. [↑](#footnote-ref-2)