A MINIMUM VIABLE PRODUCT FOR THE PARALLEL TERRACED SCAN

MICHAEL ROBERTS
michael@vivtek.com

NOVEMBER 04, 2025

Douglas Hofstadter's research group produced a dozen projects over a period of four decades exploring the mechanisms of human thought; my intent is to extract the core functionality they all share into a reusable methodology. This first project in that effort examines the *parallel terraced scan*, the architecture that allows processes to build abstract structure as the emergent result of many small and manageable agents. Existing projects present this architecture in the context of an involved doctoral work, so for greater clarity and to simplify testing in isolation, I've defined a much smaller problem domain as a test case. This report sets the background for this effort, talks about the architecture and the test domain used to implement it, and expounds on a few lessons learned during implementation.

All the code from this project is at https://codeberg.org/Vivtek/AI-TerracedScan; once I've put it through the wringer with a more detailed application, it'll hit CPAN as AI::TerracedScan.

Contents

What I think I'm doing here

This document is essentially a chatty technical report, the description of progress made in this first stage of what would have been my doctoral work had my life run on a different track. I think it's reasonable, then, to describe what that work would have been and how I now see it, and give you a sense of why it's still relevant. It is vaguely aimed at both friends and family ("You've been doing *what* now?") and at FARGonauts and other

FARG = Fluid Analogies Research Group

interested technical/academic people, and I hope it suits all those audiences. Feel free to let me know where it doesn't, and I will try to recalibrate for the next publication.

As is true for so many people who have worked with Doug Hofstadter, my first exposure to his work was GEB:EGB[1]. I bought the book and devoured it when I was in tenth grade, and when it came time to do graduate work, there was really only one place I wanted to do it. By that time, I had already worked professionally for several years in software development, and so when I had the opportunity to immerse myself in the FARG software experience, I saw the existence of several independent codebases as an opportunity to do software maintenance—to work towards reusability of a central core of functionality that would make future projects easier to manage. The year was 1994.

[1] Hofstadter, D. R. (1979) *Gödel, Escher, Bach: An Eternal Golden Braid*

But we started our family at the same time we both started graduate school, and we ran out of money about fifteen minutes in. I went back to consulting work in Indianapolis, we bought a house, I finished my Masters, and I ultimately went into database-backed web development instead of going forward with doctoral work. Then the second kid came, we bought a bigger house, we ran into health problems, and there ensued my period of wandering in the wilderness.

I made one brief attempt to resume FARG-related work in 2006. The number of completed FARG projects had risen from three to more like ten, and there hadn't been much reuse. I talked about that with some of the people active in the group at the time, and everybody agreed it was a good idea. I did some solid thinking about what such a reusable core might provide to projects, but fate did not smile on my endeavor, and my wilderness years continued.

Recently, it dawned on me that as both children are now gainfully employed and we have no mortgage payments, I no longer have any real excuse not to pick up where I left off. I started rereading dissertations, and taking stock about how I'd start doing something real. And astonishingly, I realized that every scrap of coding I'd done in support of this or that idea turned out to have been motivated by this very research idea. So what choice did I really have? It had to be done. This report represents the completion of the first step on that journey.

The overall project is simple in outline. There are, as I count them, about a dozen "official" FARG projects. Each of these projects essentially has a dual nature: first, it delineates some activity, like perceiving the structure of a sequence of numbers to predict the next few in the list, or seeing analogies between different geometrical configurations in order to define how one group of figures is different from another group. The FARG tradition calls this kind of activity a *microdomain*. The project then breaks that activity up into very small actions (*codelets*, because they're very small subroutines) that bring the system towards its overall goal, and sets up a system of applying those actions incrementally. I'll describe this approach in more detail in the next section, in the context of the earliest of these programs, Jumbo—but it's important to understand that what I've done in this first small project is to implement that *architecture*, which can then be applied to any microdomain.

But the purpose of FARG research isn't to develop software techniques—it's to explore cognitive science. So the bulk of the effort in any given FARG project is rightly devoted to the microdomain itself and what it can say about human cognition; the code that implements the process is necessarily high-quality because it has to work reliably to support the research, but like a lot of academic code, it's one-off. It's not built for maintenance or reuse, and it's generally very complex. The exception to this rule within the FARG group is Abhijit Mahabal's PySeqsee, the first FARG attempt at a common platform. There have also been a few projects outside the group that have reimplemented the basic FARG architecture with some view to reuse; Scott Bolland's FAE[2] is probably the most comprehensive. I'll be taking these works into consideration as I move forward.

PySeqsee is at https://github.com/amahabal/PySeqSee for your viewing pleasure.

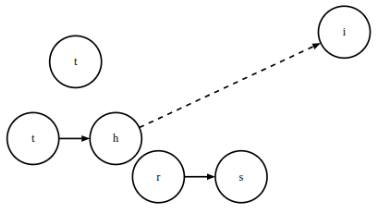
[2] Bolland, S. W. (2004) *FAE: The Fluid Analogies Engine: A Dynamic, Hybrid Model of Perception and Mental Deliberation*

Part of what I wanted to do in the 90′s—preparing the way for my *real* work of reconceptualizing the core FARG functionality on a more semantically sound basis—was to reimplement a couple of these existing microdomains in a "main sequence" of sorts (the sequence was a lot shorter in 1996), and come up with something I now envision not as a reusable "library" of FARGian code in any particular programming language, but really more as a methodology for taking the conceptual structure of a given domain and designing a codelet ecosystem that expresses it. Ultimately, this effort should end up sketching out a semantic basis for codelet design, and in later research I hope to use that semantic design directly using a low-level infrastructure of standardized codelets, so that writing your own codelets will ideally not even be necessary. (Although I can imagine that writing specific codelets is a sort of "compiler step" that could increase performance on well-understood tasks.)

When I first started planning this run at the goal, I reasoned that ontogeny should recapitulate phylogeny, so I decided to start my FARG odyssey with Jumbo, largely due simply to the fact that as the first of the FARG projects, Jumbo has the simplest architecture. It's the only example of a microdomain implementation with no conceptual memory, and I wanted to make things as simple as possible for myself as I got things off the ground. As it turned out, even Jumbo wasn't simple enough. Once I started actually coding it up, I got lost in detail, and after a couple of months I lost momentum.

My FARGyssey, if you will.

This was due to two things I should have been prepared for: I spent too much time generating nice Workspace diagrams and accrued technical debt doing so, and I didn't have a good unit testing regimen in place because I was testing the scan engine and Jumbo at the same time, but without testing either very well. I realized that I simply needed to start over from scratch without graphics and with sane testing—and, in the end, without Jumbo. I needed to design and build a minimum viable product before I could really move on to more serious domains, and that's what this report is about.



Not that the diagrams weren't fun! (And animated.)

Let me fill in a bit more detail about the history and architecture of the parallel terraced scan, and then I'll describe my MVP strategy.

Brief history of the parallel terraced scan

The concepts which led to the parallel terraced scan, as so much of Hofstadter's work, were amply prefigured in GEB:EGB in 1979, but the first publication that *named* it was his 1983 article on Jumbo[3]. The core mechanism is so simple that it's survived essentially unchanged to the present day. It's the software equivalent of the stably expressed genes that code for cellular metabolism; too much mutation and the organism just can't function.

[3] Hofstadter, D. R. (1983) *The Architecture of Jumbo*

It's not by chance that I refer to it in biological terms, either. Hofstadter took his inspiration from the action of enzymes in the cytoplasm of the cell, and conceived an architecture that harnesses the actions of many small agents to achieve a high-level goal holistically—the result we want is the *emergent* work of agents that have no particular knowledge of how they fit into the system, or even that there's a high-level goal to achieve.

The amino acids of this process are individual units of data in what is now almost always called the Workspace (it was called the Cytoplasm in Jumbo), and the agents are (ideally short) bits of code we call *codelets*. Codelets are "queued" on a list called the Coderack—the scare quotes are because the Coderack isn't really a queue; codelets are taken from it *at random*, which is one point where nondeterminism is allowed into the system.

Each codelet takes a look at the current state of the Workspace, decides whether its action is still relevant, and if so, makes some decisions and takes action in the form of changing the Workspace contents and then optionally posting some follow-up codelets. In most implementations of this architecture, individual decisions that are taken (including the selection of codelets from the rack) are "more random" if the *temperature* of the Workspace is high, and more deterministic if the temperature is low. The temperature is a metric that estimates the quality of the partial solution currently built, so when the scan "feels good" about what it's doing, it tends to follow its current line of thought more closely, and if it "feels bad" then it's more likely to explore alternatives. If the temperature gets high enough, one option is just to scrub everything (or parts of everything) and start over.

I'm being deliberately vague, by the way, about what it means to "feel good" or "feel bad". The calculation of this quality metric is very domain-dependent. In my test domain NaiveSort, I lower temperature when two numbers are bonded in a sort order, and I raise temperature whenever a number tries to sort itself into a list and fails. In this case, temperature turns out to be a dynamic metric, less an indicator of quality than an indicator of current progress and whether things might be going off the rails. At any rate, I don't really understand much about temperature calculation in general yet—that's the point of this entire undertaking, to evolve some best practices.

There's a lot of subtlety to this architecture. It's easy to see terraced scans *everywhere*, from the way humans explore research topics to the way ants move a piece of dog kibble towards the door. It's nearly impossible to see a scan in action and not think, "Hey, this Hofstadter guy was really onto something! Why don't we see this architecture used everywhere?" That's a matter for a different—and much longer—publication, less a technical report than a manifesto, perhaps even a screed.

I think in the end, though, it boils down to one reason—it's just too dang hard. I'll delve into this in more detail in my lessons-learned section, but—while the parallel terraced scan is actually surprisingly forgiving, getting reasonable results even from sloppy rulesets—thinking like a codelet ecosystem designer is *hard*, and involves both a certain species of weirdness in the programmer and a willingness to accept that the pieces of your program are lower-level than your words for talking about the problem space.

It has been used in some truly impressive work, though! In addition to the dozen post-Jumbo FARG projects, which worked with microdomains ranging from analogies to games to musical perception, I know of another handful of extra-FARG projects applying the same architecture to robotics and various aspects of language parsing and generation.

I initially shoehorned a timeline into this section, but it's too much detail. There's a fairly complete timeline of the FARG projects at https://github.com/Alex-Linhares/FARGonautica/ along with a decent bibliography, and eventually I'll put together one that explains the non-FARG projects as well.

I sort these into two basic architectural categories. The simplest tend to show up earlier in the timeline and consist of the application of the parallel terraced scan to a single perceptual activity; these were the projects for which I originally started seeing a need for a common approach and I think of them as "TS classic" systems: Jumbo, Numbo, Copycat, the Letter Spirit Examiner, Gan's Chinese parser, Tabletop, and CMattie/IDA.

In the second decade of the FARG timeline, we start to see systems that need more complexity than a single Workspace instance, and add additional memory and workflow structure to provide context for individual Workspace runs: Metacat, Letter Spirit Part Two, Phaeaco. Then, in the new millennium, we see what I think of as the "second generation" of systems, which build on this mature body of theory: Seqsee, Linguoplotter, and a few others like Musicat and George that I haven't yet read enough about to categorize.

The TS-classic systems are where I'm starting, and they almost universally add one central architectural feature to the basic parallel terraced scan: some kind of conceptual network, most commonly called a *Slipnet* from its name in Copycat. This turned out to be such an advance over the simple Jumbo terraced-scan-only architecture that something like it features in every later project. Later phases of this research effort will examine it in more detail.

A unified FARG theory, then, has a lot of ground to cover! My goal for this research effort in general is basically to develop a reasonably complete catalog of architectural features and codelet design techniques that could then be used to develop any of these projects from a single starting point, then—unreasonable as this may seem—to redevelop a representative sample to ensure that the approach works. Given their widely varied nature, I sometimes fear that the representative sample will turn out to be all of them. But ultimately, I've been convinced for thirty years that lining a bunch of these projects up and comparing them will reveal commonalities that can then be mined for a kind of unified semantics of thought.

Implementing a first domain

Until I've implemented something, I don't actually understand it. I've read a lot about the FARG projects—and since I found myself restarting my shadow doctorate, I've read a lot more. But faith without works is dead. Any theoretical listing of features might be of historical interest, but the only way to understand why a given domain might require this feature or that feature would be to implement it. Logically, the place to start would be the simplest possible domain that used the fewest architectural features.

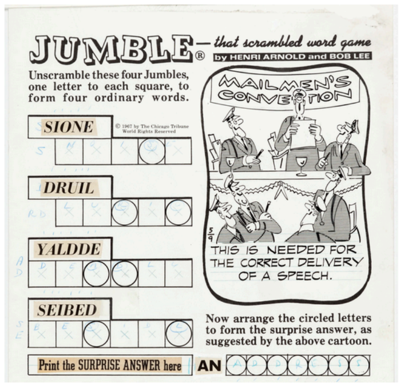
At this stage, by the way, I *only* want to implement the scan engine itself. Especially the later projects like Letter Spirit, Metacat, and Seqsee are organized on the basis of an overall architecture that uses a workspace instance to solve a specific task and then embeds that instance into a larger process. Letter Spirit actually has three different terraced-scan contexts, each operating on a specific part of a common scratchpad memory and coordinated by a higher-level deliberative process. Seqsee can spin off Subspaces to deal with local tasks in the context of the larger ongoing process. And Metacat is explicitly concerned with self-watching and tracing of individual instances that are components of a higher-level process.

This initial project can safely ignore all of that, implementing only a terraced scan instance for a particular task run and remaining agnostic about the bigger picture, although we will want features that make it easy to manage the instance and get information in and out of it.

Unsurprisingly, we will therefore be looking at the "TS classic" generation—Jumbo, Numbo, Seek-Whence, Copycat, and Tabletop. Of those, only Jumbo gets by without any conceptual network. So when I decided to take my first steps into FARG implementation, I decided Jumbo would be my first domain.

I don't want to go into too much detail in this report, but the original microdomain in which the parallel terraced scan was developed was Jumbo, and it remains the best way to present how the whole thing works. Jumbo's purpose is to model (some of) what goes on in your mind when you solve a newspaper Jumble puzzle. The Jumble is usually on the comics page, the only part of newspapers I've traditionally paid attention to, and so I was familiar with it long before I knew Hofstadter had modeled the cognitive processes used to solve it. Before, in fact, he *had* done that modeling, starting in 1982.

My grandmother would save "the funny pages" (that's Appalachian for "comics") and give me a stack when I visited. The Jumble puzzle looks like this:


JUMBLE® — that scrambled word game
by HENRI ARNOLD and BOB LEE
Unscramble these four Jumbles, one letter to each square, to form four ordinary words.
© 1967 by The Chicago Tribune World Rights Reserved
SIONE
DRUIL
YALDDE
SEIBED
MAILMEN'S CONVENTION
THIS IS NEEDED FOR THE CORRECT DELIVERY OF A SPEECH.
Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.
Print the SURPRISE ANSWER here
AN

If you've never noticed the Jumble, I've put a random example over in the margin. The point is to unscramble each of the non-words into words, then take all the circled letters and unscramble them into the punchline of the little comic. The humor in these comics will never challenge you, but the puzzle may. If you spend even a short amount of time with these, you'll probably notice that there are two levels of solving them—for short words like the first in the margin example, you'll often just flash on the solution. (It's *NOISE* if, like me, you want to cut to the chase.) But for longer strings, you might find yourself "manually" trying out different parts to see if they make sense.

This part is important to understanding what the Hofstadterian project is all about: a lot of what we model is what your brain does that you perceive as an instant flash of insight. You can push letters around in your head to stimulate (or simulate) the process, but spend five minutes with word scrambles and you'll know what I mean—you don't see what the word could possibly be, you doubt the very existence of words, and then suddenly, a word is just *there*.

Something that deserves note here, by the way—and this is something that Jumbo doesn't address at all; I'm just leaving it here as a tantalizing taste of what's to come in this larger research track—is that when a human looks at these scrambles and doesn't immediately see a solution, that's not the end of the story. After I found this example Jumble and wrote the above paragraphs, I went to peel some potatoes for lunch. As I was standing there, I flashed on the next two words: *LURID* and *DELAYED*. How does *that* happen? I have some ideas, but they won't get implemented this year, that's for sure. (Note that *DELAYED* doesn't actually work—my memory had warped the letters. You'd think that would be a flaw in how the mind works, wouldn't you? You'd be wrong, although nothing I report here and now will back that confident statement up.)

Once you have a word in mind, whether immediately or while cooking lunch, you can play around with it, read it backwards, swap letters, and find other words that might also fit into the same slot; this is part of the larger puzzle and is more deliberative, more conscious. But all that only starts once you've perceived a word in the first place.

Jumbo models both ends of this in what is really a two-phase process. It starts with a pile of letters in its working area. Little bits of logic called *codelets* then examine letters at random (we call this "sparking") and decide whether they fit together, on the basis of a list of good letter combinations called the *chunkabet*. If they do, they're "bonded" in groups of two or three. Once there are little bonded groups of letters floating around, more codelets look at those groups and decide whether to commit to them and take their letters out of play, a process called "glomming" them. I think of gloms as being little tiles with groups of letters on them, like *str* or *oi*, and once we have some glom tiles to play with, we start collecting them into syllables and chaining the syllables into a line. Once a group of letters has been chunked into a glom, the system no longer looks at the individual letters as possible additions to other groups.

None of the codelets advancing this process has any kind of overall checklist or plan, and in fact they're posted to the Coderack continually and drawn randomly one at a time to run. Each can look at a randomly selected unit of a particular type, or at a specific unit it was bound to when it was posted. It's all a seething soup of activity.

After this seething soup has glommed all the letters available (sometimes in groups of one, like the *N* in *NOISE*), we have something not guaranteed to be a word, but that kind of looks pronounceable by an English speaker. Hofstadter calls this a "pseudo-word"; I call it a *wordoid*. ("Wordoid" itself being something of a wordoid.) The second phase of the process starts now, and *more* codelets examine the wordoid's component gloms to see if they can shuffle into a better wordoid, where "better" means "higher chunkabet score"—meaning more pronounceable.

There is actually a third phase in the process—because sometimes, the first two parts get stuck. A letter might turn out to be incompatible with existing gloms, for example, yet can't stand on its own. The FARG tradition calls this kind of situation a *snag*. One strategy to deal with snags is to invoke yet another category of codelets, the *breakers*. A breaker looks for dubious bits of structure that aren't helping us, and smashes it. Not every run needs to resort to this kind of desperate measure, but when you need it, you really need it.

It is absolutely vital to realize that nowhere in any of this is any list of valid English words. Most people find this shocking and dismaying, but actually *knowing words* is a whole different process that isn't the one we're modeling here. Anybody with a week's experience programming can shuffle letters and compare the resulting string to a word list—that can be a useful tool, and it's how we'd approach something you'd want in production, like scanning for spam or doing database lookups, but it's not what we're about here. We want to capture the regularity of the English language in such a way that you can imagine the system seeing something wordlike that *nobody had seen before*. To be creative, in other words, although it's clear that this is a very low level of creativity.

When I set out, I implemented letters and sparks and bonds and codelets and realized I needed some better intuition of what was going on in there, so I wrote a neat little display that animated the sparks shooting between letters and how bonds pulled them together. In retrospect, I had bitten off too much to chew; when I started work on the glomming stage, I found myself thinking about how to animate the bond-group-into-glom transition more than I was thinking about, you know, actually building the codelets that would do the work.

One way to get past this kind of snag is to break up some of your structure and start again. I resolved to do just that—but with a smaller domain. I needed a minimum viable product for my first implementation of the parallel terraced scan.

A minimum viable product: NaiveSort

But what could be even simpler than word scrambles? Since I already had bond chains (the two-or-three-letter groups) working pretty well, I thought maybe to test with a "miniJumbo" that would just build the bond chains, with some vague idea that I would have a minimalistic chunkabet to avoid putting *that* in the standard distribution. Then, when I actually sat down to code it, it occurred to me that I could save myself a lot of trouble by coming up with a domain that didn't need a list of good combinations at all. Instead of arbitrary sequences of letters being judged for quality, why not just sort numbers?

This is my "nanodomain"—so called because it's not even big enough to be called a microdomain. It makes no pretense of exploring the mechanisms of human thought; it just applies some simple rules to sort numbers into lists. I called it "NaiveSort" because it's not even particularly great at sorting. Not to say that it sorts *incorrectly*, but it doesn't bother to do it efficiently.

The entire model consists of four codelets: *number-spark*, *spark-checker*, *done-checker*, and *bond-breaker*. The *spark-checker* codelet is posted whenever the last spark-checker has executed, so there's always a chance of sparking. It selects two numbers from the Workspace at random, and if they're not already sparking and not already bonded, it creates a spark unit linking them and posts a bond-checker follow-up codelet.

If it's easier for you to just read the code, the NaiveSort domain is defined at https://codeberg.org/Vivtek/AI-TerracedScan/src/branch/main/lib/AI/TerracedScan/Domain/NaiveSort.pm as part of the overall AI::TerracedScan module.

The *done-checker* is similarly simple. It gets posted only after there is at least one bond in the Workspace, with 20% probability, meaning it will be posted on average every five ticks. It selects a number at random, asks for its *bond neighborhood*, and if that neighborhood is the same size as the Workspace (meaning that the group this number belongs to is in fact *all of the numbers*, meaning the whole Workspace has now been sorted), it posts the

The Workspace can find the neighborhood of any unit by following links of whatever types are appropriate. This is a neighborhood, and it's the kind of context that makes codelet design so much easier to comprehend.

response. Once a response has been posted, the scan halts with that result.

At first, I didn't write a *bond-breaker* at all; I hadn't quite convinced myself it would be necessary. I'll talk about that in a minute, but first let's look at the codelet that actually *does* things, i.e. actually bonds numbers into sorted pairs, then lists, then eventually one list with all the numbers in the Workspace.

Feel free to skip this paragraph and the next if you find them boring. They're just implementation details.

The *spark-checker* is considerably longer than the other codelets. Starting from a spark unit, it gets a bond chain summary from the Workspace for its two contained numbers. The bond chain summary consists of the bond neighborhood for each of the units, along with some initial tests: it checks whether their bond neighborhoods are the same and sets a "bonded" flag if so, and it also sets a flag for each of the two units: "u" if the unit is unbound, "s" if the unit is bound to only one other number (and is therefore at the end of a bond chain), and "d" if the unit is bound to two other numbers (and is thus in the middle of a chain).

Based on that, the spark-checker can categorize the potential bond as *uu*, *su*, *du*, *ss*, *sd*, or *dd* based on the situations of each of its two units. The rest of the codelet handles each of these cases individually: *uu* is always bonded and *dd* always fails. *su* succeeds if the unbound unit would extend the appropriate end of the bond chain of the singly-bound unit. *du* succeeds if the unbound unit fits between the doubly-bound unit and its neighbor on either side. *ss* succeeds if the two bond chains can be merged; it therefore fails if the two singly-bound units are both at the top or bottom of a chain or if the chains would overlap. In the *sd* case, if the singly-bound unit fits between the doubly-bound unit and one of its neighbors, then the singly-bound unit leaves its existing bond chain and merges into the other chain.

But stop skipping here—this is important!

I chose these rules because they're manifestly crazy. If you were really sorting numbers and you had two sublists to merge, say *1-2-4* and *3-5-6*, you'd instantly notice that they're *almost* sorted together, and you'd say, oh, *1-2-3-4-5-6*, done! NaiveSort just says, "These sublists overlap and so I don't know what to do," and *bond-checker* simply fails. I wanted rules that were quick and easy to write without thinking of a lot of special cases, and frankly I wanted rules that would just sometimes fail to work, to see what would happen. I figured there would surely be some cases where these rules would "get stuck" and be unable to figure out a way forward.

This is where Doug Hofstadter and I differ. He would spend a minute thinking up specific lists and would already have worked out a dozen cases where these rules would lead to breakdowns, then he'd write the software and see that it did exactly what he expected. I am not like that. I went out and mowed the road and poured some leveling concrete on the back terrace, then I wrote the software and it did things I *didn't* expect, not in detail. I had in the meantime already seen that a two-number bond chain would sometimes be unable to break itself by the rules I'd given it, and so my test case had seven numbers, to ensure that the Workspace would never consist only of two-number bond chains.

You may already thought of one case where my stated rules would lead to a snag, even though I hadn't. My test list was *1, 3, 4, 6, 7, 10, 11*. About the third time I ran a free run with that list, it didn't return. Ever the optimist, I hadn't written any abort code that would handle the situation if the scan got stuck. Once I did that, I quickly found that the system had identified a Workspace configuration that it could never break out of: *1-11*, *3-4-6-7-10*. Neither of the numbers in the two-number chain could ever fit within one of the bonds of the longer chain, and that's the only way the two-number chain could ever be merged into the other chain. The long chain couldn't merge into the two-number one because a two-number chain has no doubly-bound unit to merge with (the only merge case I'd given it).

Fairly quickly, the system provided more instances of configurations that could never resolve, including one with an unbound unit! (This surprised me because I thought the rules for unbound units would always make sure it was bound.) A couple of examples are *1-10*, *3-11*, *4-6-7* and *1-7*, *3-11*, *4-10*, *6*. It was an absolute delight to see my code discover them.

But it brought home to me the fact that sometimes, instead of building structure, we really do have to break existing structure down if it's causing problems. A number list sorter that just throws up its hands and aborts isn't very much like people—or rather, it's like a person who hasn't learned how to sort lists and doesn't think it's worth their time to figure it out. Part of figuring things out is to try to look at things in a different way, and one way to do that is just to break some of the bonds in the Workspace and then get back on track.

Another way to figure things out is to learn new codelets, but that's not happening this year.

I didn't want bonds to just spontaneously break all the time, though. It would *work* (I checked, by implementing it and watching it work) but it seems inefficient to break up perfectly good bonds when things are going well. This intuitive feeling that "we don't want to interrupt progress when things are going well" is generally modeled in FARG models by a computational temperature, which is low when things are "looking good" and high when things are "looking bad". I'd been eyeing this concept with considerable mistrust—how do you know when things "look good"? All I knew for sure about NaiveSort is that things were good when there were more bonds, but that wouldn't be a practical way to know when to break bonds—after all, when there are *no* bonds, that would make it harder to make any. That couldn't be right!

This would be a good thing to run statistical tests on.

I eventually decided to let codelet action figure it out for me. I introduced a "frustration" measure to the Workspace, and incremented the frustration of each number when it sparked but the spark failed to bond it. And when a bond *was* formed, I set the frustration of the entire new chain back to zero. This is pretty hamfisted, like everything in NaiveSort, but it did allow me to take the total frustration of the Workspace. By combining the number of bonds with total frustration, I came up with a temperature that seemed to kind of reflect the notion of things going badly when the rules hit an impasse. And once that number gets high enough, the system starts to release *bond-breaker* codelets that select a unit at random and break one of its bonds.

Once I'd done that, NaiveSort never aborted. When it reached one of its "bad configurations", the failing bonds would slowly raise the temperature until it hit my arbitrary trigger of 75, a bond codelet or two would be released, and once those newly-free numbers bonded the temperature would quickly fall again. They often would have bonded elsewhere, and then NaiveSort would terminate with a sorted list. Sometimes it stumbles through several bad configurations before getting on track, but the breaker mechanism rescues it every time.

I was honestly somewhat astonished. This entire process had taken me only about a week (the breakers took a little longer). Thirty years after first formulating this plan, I had finally written a terraced-scan domain, even if a half-witted one. But what really bowled me over is that a half-witted set of codelets *still work*. This Hofstadter guy was really onto something!

Features I consider minimal

That gives us the background to understand the "standard FARG" feature set that I needed to implement this minimal domain. I did come up with one novel feature and one marginally novel feature that I couldn't force myself to part with, but I'll talk about those in the next section. Everything here is just a straightforward implementation of pretty standard FARG fare.

First and most obvious are **the Workspace and the network of units it contains.** These are the entire point of the scan. Units come in two basic flavors, although they can be a mixture as well; by default, a unit either contains an item of data (I roughly consider these to be percepts; they're the input elements of the task to be done) or a link between two other units, which I also speak of as "containing" the other units. Each unit has a type, which is a string. In NaiveSort, units are numbers, sparks, or bonds.

The Workspace is a table that tracks all current units by a scalar ID, as well as a few status markers. The default is a single marker "dead", but others can be defined in a given Workspace instance. The Workspace also tracks the population of each type (so it's very easy to see if there are any bonds in play, for instance) and provides quick ways of selecting units at random.

Dead units are those that have been killed during the process. Units are never deleted from an active Workspace, because the Workspace is intended to be a task-specific single-use structure. Eventually I'll want to implement it in a fast, low-level language and won't want to waste time on garbage collection in a structure that isn't going to last anyway. Moreover, codelets could explicitly search for dead units as a sort of augmentation of short-term memory: if a given bond has been tried before with no chance of success, why try again? In the context of a larger cognitive process, I envision the Workspace as being assembled from long-term memory, running on a specific task, and then forming a new episodic memory trace before being cleaned up. Dead units won't be remembered; that's the extent of garbage collection in this system.

Unit containment. Units can contain one another, and this is a bidirectional relationship, meaning that a unit knows who it contains as well as who contains it. Units also have types. Unit types can be used to control codelet release and targeting, and the unit type can change over time (which I call "promotion", although my original reason for that name involved a hierarchical model that I no longer use). This means that the unit's type can be regarded as a state machine and that it can attract different codelets—or no codelets—over the course of its lifetime. In combination with containment, the result is that containers can *chunk* their contents if they are promoted to a type with no codelet associations, removing them from active processing. Promotion is used in NaiveSort only to turn a spark into a bond if the bond-checker succeeds, but Jumbo will promote letters to "glommed-letters" once the bond group is fused into a glom, so they're no longer available for consideration in the bonding process.

The Coderack and codelet set in play for the current run. The Coderack maintains a table of current codelet instances and a second table, its enactment, which is a list of all the codelets that have been run and their outcomes. A codelet *instance* is a bit of code along with the specific parameters (generally a unit or two) it will run on. The instance, when selected and run, can either fire, fizzle, or fail, which correspond to taking appropriate action, recognizing that its action is no longer appropriate in the current context, or failing the criteria for taking action. Each of these outcomes is recorded in the enactment along with the "rule" that was applied; the rule is simply a short text that denotes which decision branch led to the outcome, for debugging purposes.

Codelet posting routes. A codelet can (currently) be posted in three ways. The first is simply an explicit post from outside the scan. Codelets can also post follow-ups; all follow-up codelets keep track of their invocation origin, again to simplify debugging. Finally, scout codelets are posted at regular intervals based on the current population of unit types in the Workspace.

This minimal terraced scan doesn't include a Slipnet or anything like it, but the unit type populations can be seen as a very thin analog of the Slipnet's ability to post conceptually appropriate top-down scouts. In later phases, this mechanism will be used by the conceptual network to do that, almost certainly with some modification from the current state of the code.

Temperature and decision bias. Nearly all implementations feature a global temperature that regulates the determinism of the process; this temperature is then varied as the quality of the potential solution changes. This description strikes fear into my heart—while it's difficult to design codelets, I have even less idea how to calculate the global quality of the Workspace. This seems to embody considerable domain knowledge that can only be discovered over the course of development. Fortunately, even my hamfisted temperature calculator in NaiveSort seems to have done what it needs to do, so presumably we can get away with a lot of handwaving in this area. Copycat calculates temperature based on structural aspects that I don't yet understand, so it might be possible to come up with some principles for temperature calculation design later.

Features you might not consider minimal

There are a handful of features or concepts that I included in this initial project despite all reasonable effort to avoid novelty.

Framelets. I found it difficult to write good codelets. I'll be talking about that a little more in the lessons-learned section, but I landed on a general strategy of breaking each codelet into two phases: the first phase looks around the Workspace and establishes some context, and the second phase applies some rules and decision logic to that context in order to fire, fizzle, or fail. In simple cases, this seemed easy, but in more complex cases it got progressively harder to organize the context part.

Just to give this tension its proper background, though, I should explain that the reason I want a unified Workspace execution engine in the first place (in other words, my motivation for undertaking this project) is to support my ultimate goal of implementing Langacker's cognitive grammar. At some point, I came to the realization that this contextual snippet of the Workspace, which I originally thought of as the "neighborhood" of some target unit (for instance, the neighborhood of a letter "c" unit that has been bonded to "r" in a Jumbo run is the bond group of "c", "r", and the linking unit) was actually more like a *frame*. But it was a *little* frame. Late at night on May 29, 2024, I coined the term "framelet" to describe it, and this coinage sparked a descent into madness as it threatened to realign my entire understand-

This was my goal in 1996, anyway; in later years I generalized that to the intent to "implement human semantics," whatever that ultimately proves to mean.

ing of what the Workspace even *is*—indeed, I realized that Copycat's Slipnet can be seen as the emergent snapshot of a cloud of "ghost units" retrieved from lexical memory and implemented as framelets, the expectations of which give rise to conceptual pressure to recontextualize the contents of the Workspace.

I have to admit I am quite taken with the idea of ghost units exerting spooky pressure on the Workspace.

It took months of doing other things for me to regain the clarity to implement the Workspace without all that *thinking* going on, but the neighborhood context extractor still has to return a framelet to make sense. And in fact, the calm and minimal framelet module I ended up implementing gives us some very convenient tools for writing codelets—it groups a few units, but it can also mark some of them as special, as what Langacker might call foreground or trajectors. This made codelet design a lot easier, and so the framelet concept remains fairly central to my implementation. You wouldn't *have* to use it to write effective codelets, but I certainly will.

Frustration. The other feature I explored and included in NaiveSort was the idea of frustration. This is an indexed status of the unit, meaning Workspace queries return it directly, and it is incremented and decremented during codelet actions. Specifically, it's incremented every time a codelet fails on a unit, and it's cleared for the entire bond chain every time a unit is added to the chain. (I'll talk about bond chains later, but they're exactly what they sound like.) So this isn't really a feature of the Workspace so much as it's afforded by it—the codelets handle it.

Right now, I've just used frustration as one component of global temperature, but later I may implement a system of biased selection, so that sparks can preferentially focus on particularly frustrated units. This can also be seen as a sort of "local temperature", so that the Workspace ends up with a heat map. NaiveSort works fine without that biased selection feature, though, and so I've saved it for later.

Event subscription. I vaguely remember sitting in on a seminar series or possibly even a class in 2006 during my second, grazing orbit of FARG. No notes of this survive, so my sodden meat memories have to suffice. At any rate, during that time is when I realized that there had to be a strict separation of concerns between the parallel terraced scan itself and any interactive presentation of progress. The solution I envisioned then, and implemented now, was an event notification mechanism.

Each event that changes the Workspace (adding or killing a unit, promotion of a unit to a different type, and periodic free-form status updates) can be published to any number of subscribers. This can be used for display or (later) could be used to implement Jim Marshall's Temporal Trace for self-watching, and it can be logged for later display playback or whatever analysis you care to perform. The idea here is to provide a clean break between actual execution and—especially—whatever status dashboard might be provided in the execution environment.

Lessons learned

Even this brief dalliance with FARGware programming has left me with more insights about the process than I expected. The total coding time over the space of a month couldn't have been more than five days, with another week and a half spent mopping up details like better testing and a bulk test run organizer, and I fully expected such a trivial domain to be relatively boring. It wasn't, though.

Codelet ecosystem design is hard. When I finally got the basic engine doing what I needed it to do and started debugging the actual codelets—even though there were only three—I was surprised both by how the game felt different and how difficult it was to keep things straight. I alluded to this a couple of pages ago, but it's surprising how much of the actual goal behavior of the system is emergent. I kept thinking I needed some explicit code or data structure that tracked bond chains (the sorted subgroups of numbers)—but they're unnecessary! And in fact adding something like a bond-chain unit that would keep track of each chain would be a lot of work to get right; you'd have to make sure it was notified when a unit was added to the chain or removed, and if a two-number chain is dissolved then, what, there's no more bond chain unit? It would be a mess to keep track of!

So you just … don't. This set of codelets works with numbers and bonds, and that's it. It's true that the bond chain summary extractor does know how to follow bond chains, but I suspect you could get away without even that much knowledge of the domain built into the code.

Framelets helped. Your mileage may vary, but starting off by asking "what does this codelet need to know about the Workspace context?" really made the rest of the logic easy to streamline into a sort of decision tree. My original goal when rebooting this initial project was not to introduce any new constructs at all, but I couldn't resist the framelet concept. I expect both context extractors and most codelets to even out into a set of standard types as I implement more domains, but we'll see how right I am about that.

A lot of knowledge about the domain isn't reflected in the code. And that's a real problem. If the purpose of your code is emergent, then *nothing in your code documents its purpose*. In the brief time I spent thinking about bond chains and how they could merge and so on, I got the barest taste of the immersion that any FARG doctoral candidate working on a domain for a few years must necessarily experience. In any kind of deep immersion in a topic, the human mind responds by proliferating distinctions and jargon that may never have been imagined previously. Semantics just zooms in—*this is exactly what makes AI hard.* So did any of my deeply meaningful insights make it into the codebase? No, obviously not, because the codelets don't even know they're building bond chains. There are two reasons this was a disappointing discovery.

There's a relevant XKCD for this (as for everything), but I can't remember the right keywords to find it.

First and more prosaically, if someone needed to maintain this code—or, let me assure you, if I myself needed to maintain it next year—unless I wrote these insights down and documented the mental shorthand and jargon I came up with just thinking about something as trivial as bond chains, all that would have to be rediscovered even to understand what was going on in the system. I've been working with software for (checks notes) forty years now, and I still don't write everything down that I need to understand even simple code, let alone an AI research project. I keep a development log for everything I code—but even if I manage to express some of these insights, there's no guarantee I can find that information later. Like this report, my devlogs are wordy, but unlike this report, I don't edit them for clarity.

Of course, this is true of any software, to a certain extent—it's one of the key challenges in software maintenance. But I felt it more keenly in this context, due to the second reason for disappointment. In the context of what I hope to achieve with a FARG-based approach, *my* goal is to approach semantics. If the semantic insights I developed while writing the domain code didn't make it into the code at all, then have I missed the point? I think I haven't, actually, although I needed to think about it a little.

The Workspace and the task of perception it is carrying out, after all, are not a full model of cognition. The focus of the FARG model is to work at the level just below our accessible thoughts —and that means that deliberative thought is not even supposed to be located here. It seems reasonable to me that even if the Workspace process has no knowledge of higher-level concepts like bond chains, a self-watching module *could*. So the codelet ecosystem may have no notion of bond chains or the difficulty of two-unit bond chains in the NaiveSort domain, but a watcher could easily say, "Aha, there's one of those two-unit bond chains again," and take some action or at least just know why this run took longer. The FARG literature usually calls this the "cognitive/subcognitive" distinction, but I'm increasingly thinking of it as a "semantic/subsemantic" distinction—which might just be a different word for the same thing, but again, I'm going to need more experience here to have a more informed idea.

Incidentally, two days after I wrote the previous paragraph, my reading of John Rehling's dissertation[4] reached section 5.2.4, where *he* defines the parallel terraced scan, and makes essentially the same point about subcognitive and cognitive levels, and cites Jim Marshall's dissertation on Metacat, which I have read, although it's been a year and a half since my last reading. So it's not that novel an insight!

[4] Rehling, J. A. (2001) *Letter Spirit (Part Two): Modeling Creativity in a Visual Domain*

Self-watching. A week after writing *that* paragraph, I thought it might be a good idea to go back to Marshall[5] after all. To my surprise, when I opened the PDF on my reading tablet, it was at the section on self-watching. Clearly I'd read the section at least twice, but now I have the conceptual depth to understand it. (Remember how I said I don't understand anything until I've implemented it? This is what I mean.)

[5] Marshall, J. B. (1999) *Metacat: A Self-Watching Cognitive Architecture for Analogy-Making and High-Level Perception*

The parallel terraced scan is only a healthy part of this nutritious breakfast. A full cognitive model needs to know what it's doing—*literally*. Self-watching is one way that happens, and it's at the self-watching level that we can start to talk about semantics, as the framework for seeing meaning in what we're doing. I've always known that Metacat "did self-watching," but only now do I have a real understanding of *why*. I suspect no semantic model of any complexity—perhaps no semantic model at all—will be able to work without a self-watching architecture to run it on.

The Workspace is remarkably forgiving. Even though I've been familiar with FARGware for essentially my entire adult life, I was still amazed to see that I was able to slap together a few poorly-considered rules and the system then managed to sort numbers into lists just fine. Back in the 90′s (although the Wayback only archived it in 2003), I wrote, "It is difficult to watch a Copycat run without seeing the program as a (semi-)intelligent being, and rooting for it to see this or that solution to the problem." That sense is still there. Watching NaiveSort fumble its way through a number list was fulfilling.

Now what?

When I first decided to get back into this research, I gave myself one rule above all else: I wasn't going to write anything on it until I had table stakes—until, in other words, I'd actually implemented something. I'm honestly torn whether this is sufficient table stakes or not. It's been an almost trivial step towards what I want to be doing. On the other hand, I also told myself that once I *did* implement something, I was going to have to write about it or it wouldn't count. I hadn't realized just how incompatible these two rules were, but as we've established, the Workspace is forgiving. I don't know if this is table stakes or not, but I've written about it.

Next, obviously, I'll finish implementing Jumbo. I expect this to include a letter-and-bond system that creates candidate bond chains (that part already works), then a glom-spark-to-glom system that turns letter bond chains (and some single but frustrated letters) into glommed tiles. I'm currently thinking I might try a more complex unit structure for turning the glom-tiles into syllables and wordoids, then implement some entropy-preserving swapping and shuffling codelets to try to improve overall quality, and then I'll stick a forkerism in it and call it done, and be *sure* of its table-stakes nature.

After that, I'll try Numbo. The Numbo system incorporates a conceptual network, which should make it easier to think about the Slipnet when I tackle Copycat. Numbo also has the distinct advantage that I have three versions of an existing codebase to work from (Alex Linhares reimplemented Daniel Defays's original code as *his* FARG warmup, and Scott Bolland implemented Numbo as one of the test domains for FAE).

Then, yes, I'll see if I can manage Copycat and Tabletop. I very much doubt that much of this tentative plan is going to survive contact with the enemy, but it gives me some direction. Along the way, I want to address a few things in a *lot* more detail.

Semantics, semantics, semantics. My larger roadmap actually consists of multiple tiers of research in what I think of as the "research stack." Each tier can roughly be seen as the infrastructure for the next tier up. This "terraced scan classic" is Tier 1 in that loose system, and Tier 2 consists of the application of the terraced scan to handle, and to take advantage of, semantic structure.

I am certainly not foolish enough to tell you that the stack has a total of six tiers. That would be madness!

Tier 2 includes self-watching, episodic and lexical and semantic long-term memory with progressive recall, the use of framelets and expectations to exert pressure on the terraced scan, and also the description of each domain at a level that allows standard tools and standard(ish) codelets to be used to implement it. I've long thought there wasn't much point in putting too much effort into thinking about that level before I had a working terraced scan and a couple of implemented domains to generalize from—but now I do have a working terraced scan.

Ultimately, the development of a new domain of application should look a lot less like writing codelets and a lot more like defining terms in natural language, maybe drawing some diagrams. And then we'll have the infrastructure to start talking about learning.

Those Workspace display diagrams. Speaking of diagrams, it's true that I embarked on diagramming too early for it to be sustainable. On the other hand, seeing the first phase of Jumbo codelets working as an animated sequence was immensely useful for developing some intuitive understanding for the process. I want to get back to that point, but on a much sounder theoretical basis. One of the palate-cleansing activities I engaged in while trying to get my plans clear for a second run at implementation of the terraced scan was a close examination of semantics on the basis of the NSM[6]. That involved a lot of diagramming, and it brought home two things to me: diagramming tools are critical for understanding complex structures and processes, and

[6] Goddard, C. (2021) *Minimal Languages in Action*

diagrams are another form of semantic presentation—one based, in point of fact, on analogies drawn between abstract concepts on one end and geometric shapes and lines on the other.

From that insight, it's a short step to the realization that a diagram of the Workspace isn't, as I had mistakenly thought, part of the target domain at all. It works from a descriptive semantics *of the Workspace*, meaning that diagramming can be seen as closely related to self-watching. As always seems to happen when I start doing this research, everything turns out to depend on everything else. But at least I can keep reminding myself that *one* part of it's working now.

Other perceptual tasks. I think it might be the case that Jumbo gets away without the need for a conceptual network because it's a relatively low-level perceptual task. The letters and glom tiles just don't have a lot of semanticity; they're tokens we can shuffle around at will. I'm fairly sure that other more perceptual, less analogical tasks like vision might be at this same level of complexity—Phaeaco draws an explicit distinction between a retinal level of processing and a higher cognitive level, and I suspect the retinal level is at the architectural level I've just developed here. So I'd like to start exploring diagram understanding as well as diagram generation.

Self-watching. Having finally understood how critical self-watching is to the overall cognitive project, I find myself thinking about it in detail this week. Over the next month or so, I'll probably work out some initial project for getting a basic understanding of how it might fit into one domain or another.

Performance. My goal in this first approximation has just been to get a terraced scan working at all. For me, that's easier in Perl and it's also easier with some of the support code I've been using for years now. None of that is written for speed; it's written to be easy for me to work with. So at some point relatively soon, it'll be a reasonable next step to move some of this functionality into C for portability and speed. I've been collecting building blocks for that for many years, so it should be more or less a matter of bolting a few of them together. Once you have something written in C, you're halfway to anywhere.

The ARC-AGI benchmark.[7] I don't know about you, but when I first saw the ARC-AGI problem set, it seemed almost to be written with FARG tools in mind. It's a set of geometrical trans-

[7] Chollet, F. (2019) *On the Measure of Intelligence*

formation analogies—it's practically just Copycat problems with a Bongard sauce. I think the place to start is by exploring the semantics of transformation rules (Copycat/Metacat) in combination with the low-level geometrical perception I just mentioned above. Should be a snap.

And with that, I'll conclude. Watch this space for further details.

References

[1] D. R. Hofstadter, *Gödel, Escher, Bach: an Eternal Golden Braid*. New York, NY: Basic Books, 1979.

[2] S. W. Bolland, *FAE: The Fluid Analogies Engine: A Dynamic, Hybrid Model of Perception and Mental Deliberation*. The University of Queensland, 2004.

[3] D. R. Hofstadter, "The architecture of Jumbo," in *Proceedings of the International Machine Learning Workshop, June 22-24, 1983, Allerton House, Monticello, Illinois*, 1983.

[4] J. A. Rehling, "Letter Spirit (Part Two): Modeling Creativity in a Visual Domain," 2001.

[5] J. B. Marshall, "Metacat: A Self-Watching Cognitive Architecture for Analogy-Making and High-Level Perception," 1999.

[6] C. Goddard, *Minimal Languages in Action*. Cham, Switzerland: Palgrave Macmillan, 2021.

[7] F. Chollet, "On the Measure of Intelligence," 2019.