Taking (Computer) Architecture Seriously

Ian Horswill
MIT AI Laboratory
545 Technology Square
Cambridge, MA 02139
ian@ai.mit.edu

Abstract

There seems to be little consensus on what an agent architecture is. In this paper I will talk about architectures in computer design, and programming languages, and what their distinctions are. I will then discuss agent architectures relative to that distinction. Finally, I will discuss recent work I have done on building better computer architectures for agency.

One of the problems with writing about agent architectures is that I don't have a good sense of what an agent architectures are supposed to be, much less how to evaluate them or what the right one is. In this position paper, I'll compare agent architectures to computer architectures and programming languages, which I feel I understand somewhat better.

Computer architectures and programming languages

Computer architecture means either the computer's instruction set or how that instruction set is implemented by a fixed collection of finite-state circuits with fixed data paths. In designing an architecture, the goal is to find simplest possible design that is still Turing complete and that has the best possible average case performance on typical programs, whatever "typical" might mean. Programming language design, by contrast, is how you specify computations, independent of what primitive circuits are used to carry them out. In programming language design, the goal is to come up with a set of features that allow typical programs to be specified as concisely as possible, yet which is still efficiently mappable onto the underlying architecture's instruction set, again, in the "typical" case.

Computer architectures and languages are both problem-independent, at least partially. Most architectures can run any program and most languages can describe any algorithm. When you switch from one program to another, the architecture is a *mechanism* that remains the same, and the language is a *means of description* that remains the same.

Computer architecture and language design, as practices, are driven by the same pair of complementary pressures to add and remove features. Adding a new feature can dramatically speed up or simplify those programs that can make use of it. On the other hand, it may uniformly slow down other programs by introducing additional overhead, or stretching the necessary length of a clock cycle. Adding GC and run-time type checking to a language greatly improves some things, but in most cases also imposes some level of overhead on all programs, regardless of whether they need or use the new features. Computer architects and language designers search feature space for the feature combination that maximizes average performance.

Average performance, however, can only be defined relative to a particular mix of programs and their frequency of use. Experimental architecture work is therefore often based on explicitly choosing a set of "representative" programs, simulating them to gather statistical data about the relative frequency with which they perform different types of operations or could use certain architectural features, and then using that data in the design and evaluation of architectures. The chosen architecture is highly sensitive to the choice of programs: a set of benchmarks based on Unix programs and a set of benchmarks based on symbolic programs will give you radically different recommendations on whether to build a SPARC or a Lisp machine.

To summarize: a computer architecture or programming language: (1) makes some things easy to do and makes other things hard to do; (2) draws its efficiency as much from its selection of features *not to provide* as to provide; and (3) can typically only be evaluated relative some agreed-upon mix of programs.

Agent architectures

What then, is an agent architecture? I don't know what it ought to be, but it seems pretty clear to me that it is used in the literature to mean a variety of different things. Suppose we transfer the architecture/language distinction made above to agents. An *agent language* would then be the means of description of the tasks or techniques used to solve them, one that remained the

same when retargeting an agent from one task to another. An *agent architecture*, on the other hand, would be the actual mechanisms that remained from task to task.

Given this distinction, most agent architectures are more like agent languages than architectures. The subsumption architecture [2] clearly isn't an architecture in this sense — there is no specific mechanism that all subsumption robots have in common. Rather, they have a *class* of mechanisms in common: all are built from finite-state machines inhibiting one another, albeit different finite state machines.

Three-level architectures [4][3], or "TLAs" are becoming very popular. They involve planners stacked on top of RAPs or subsumption-like systems stacked in turn on top of continuous-time feedback systems. These clearly aren't purely programming languages, since the planner and perhaps the RAPs executive are shared from task to task, but the lower levels can change radically or even completely from task to task.

SOAR [7] commits you to a fair amount — there have to be long- and short-term stores, a production rule matcher, and all the mechanism necessary for keeping track of subgoaling and chunking, but everything else is up for grabs. It would be interesting to compare the amount of code it takes to write the production matcher to the amount you have to add to it to get it to solve your task.

I don't know that one need necessarily commit to these definitions of "agent architecture" and "agent language," I just find them useful ways of thinking about these systems. If we were to use for agent architectures something like the methodology discussed for computer architectures and programming languages, we would want to ask the following questions about our agent architectures:

1. What features does the architecture provide?
2. What features does it *not* provide?
3. What kinds of tasks do those features make easy or hard to solve efficiently?
4. What tasks are "typical"?
5. How does the architecture perform on typical tasks?

What's interesting is that no one ever talks about 2 or 4. That's not peculiar to AI; I've also heard the complaint made about computer architecture. But it's odd because the real power of architectures often comes from what they *don't* let you do. Subsumption programs don't select actions in constant time because subsumption gives you more powerful primitives than Lisp, but because subsumption gives you weaker ones. It's just damn hard to write anything that doesn't run in constant time. If you can solve the problem at all in the language, your solution will probably choose actions in constant time.[1] Of course, subsumption also has positive features such as parallelism and (then) novel means of combination (layering via suppression and inhibition). But I personally find its limitations much more interesting.

No one ever explicitly talks about what they take as typical tasks either. Clearly, much of the debate between reactive and deliberative systems was as much a debate about the relative frequency and importance of *tasks* as about architectures. Phil Agre and I once did a system that played short-order cook in a simulated kitchen [1]. Someone once told me that it was uninteresting because its domain had almost no subgoal interactions. That was in fact, precisely one of the points Phil and I wanted to make — that often times the world is dramatically easier that then worst case. This person, however, asserted that "real" tasks had many subgoal interactions and resource constraints. So far as I could tell, he didn't believe kitchen simulation was oversimplified, just that kitchens weren't "real" domains. Apart from this point, we seemed to be in total agreement.

Agent architecture as computer architecture

I am personally interested in treating agent architecture as being just computer architecture: the search for a fixed collection of finite-state components with fixed connections that can jointly perform a broad range of tasks with minimal modification. This actually rules out a number of things that we as lisp programmers hold near and dear to our hearts:

- dynamic parallelism (*i.e.* forking and joining)
- dynamic communication
- pointers
- dynamic trees
- garbage collection

This doesn't mean you can't have dynamic parallelism in your agent, merely that the architecture can't presuppose it — it must implement it. Dynamic trees require pointers and GC; Pointers require routing networks to dereference them and arbitration networks to cope with routing contention in parallel machines; These are things one does not take lightly.

The reason I'm interested in this approach is roughly the same as for the connectionists [8] — I'm interested in humans and while global routing networks are a pain to implement in silicon and copper, they're a *huge* pain to implement in neurons with any reasonable speed.

One's first instinct might be to think that nothing intelligent could be done without trees, graphs, and GC. Nearly all of KR is built on top of tree- and graph-based representations. However, it turns out that you can do a lot more than you would expect. The world models of classical AI systems are just big databases of facts. The reasoning system understands the state of the world by performing queries on the database. A perceptual

[1] Unless you simulate a lisp machine using finite state machines, but no one actually does that.

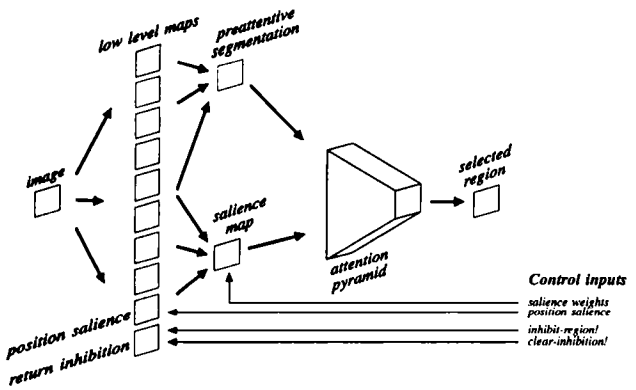

Figure 1: Structure of the attention system

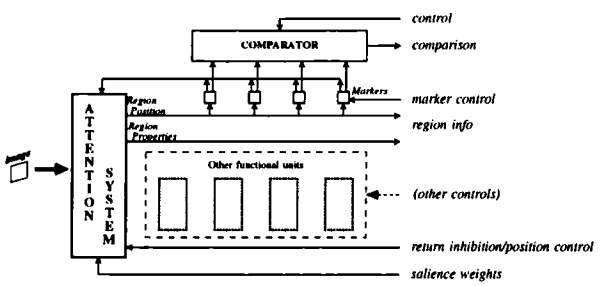

Figure 2: Visual routine processor architecture.

system that supported the same queries could be used as a plug-compatible replacement for the database.

Bertrand [6] is a primitive logic programming system based on this idea. It has a working, real-time vision system that takes the place of a KR database. The vision system computes a set of low level feature maps (color, edges, *etc*) and allows task-salient regions of the image to be addressed by specifying feature weightings (see figure 1). An *attention pyramid* finds the most salient region, based on the current weighting and routes its location, size and low level feature values to a *visual routine processor* (VRP) [9] (see figure 2). The VRP is effectively a co-processor for visual search. It provides a set of task-specific registers, called "markers," for storing image locations and an "instruction set" of useful operations on the image and the markers. The VRP can serially enumerate image regions satisfying certain feature values or having certain spatial relations to other regions designated by marker values.

For example, one can find a blue region on a red region by serially enumerating the blue regions, binding each in turn to some marker, then serially enumerating the regions under the marked regions, again binding them to some marker, and finally, testing the marked region for redness. This implements a chronological backtracking strategy similar to a prolog engine, and it can be shown that a wide range of Horn clauses can be solved by a VRP with only the addition of a shift-

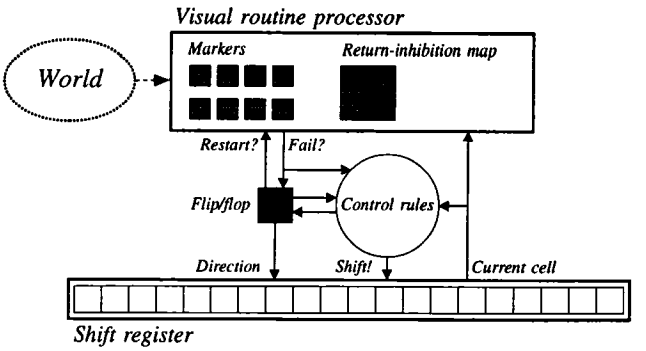

Figure 3: Solving Horn clauses with a VRP and a shift register.

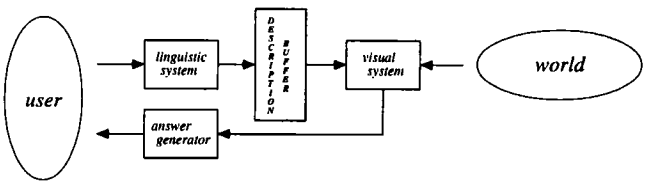

Figure 4: Top-level architecture of Ludwig.

register and a 2-state DFA to control it [6] (see figure 3). The shift-register holds an encoded form of the clause, with each cell holding one VRP operation. The DFA feeds the shift-register instructions to the VRP and shifts either left or right, depending on the success or failure of the instruction. The result is a simple system which is in many ways plug-compatible with a real logic programming database. When running on a 50MHz DSP board, the VRP processes instructions at about 10Hz.

Ludwig [5] is a natural language question answering system that uses the same visual routine processor and search techniques as Bertrand, but accepts input in the form of natural language utterances such as "is there a green block on a red block" or "is it big?" Ludwig is intended to eventually be able to do all or most of the things that SHRDLU [10] did, only in a real world, with real blocks and real visual data (see figure 5 for an example of Ludwig in action). At present, it is primitive, however. All spatial relations are dealt with in 2D, no shape matching is performed, and only yes/no questions are handled.

Ludwig is unusual in that it is implemented entirely in finite state machines described in a simple register-transfer language (although the VRP is written in C for speed). Word semantics are mapped directly into VRP operations that are stored in shift register cells. Parsing is performed by finite state machines whose job it is to signal top-level phrase boundaries (which mean the semantic system should switch to a new shift register), rather than to build complete descriptions of syntac-

tic constituency and relations. The visual system runs asynchronously and searches for satisfying variable assignments of the clauses in shift registers as it notices their completion. This allows it to find the referents of visual NPs and to test them for specific visual properties.

Ludwig contains no global pointers or routers, has no tree or graph-structured representations, and performs no GC. Yet it has fully compositional semantics and handles syntactic constructs such as nested PPs.

Like Bertrand, Ludwig implements variables as visual markers and stores Horn clauses (the logical forms of noun phrases and PPs) in shift registers. Markers and shift registers are allocated on a least-recently-used basis and can be dynamically assigned tags corresponding to high-level semantic roles such as "theme." Ludwig is able to do things that would normally require trees and full pointers using only a few relatively cheap constructs such as associative tagging and image-plane markers. By using simple constructs and pipelining its processing as much as possible, Ludwig is able to process a word in less than a dozen gate delays. A typical von-Neumann uniprocessor would require 20 gate delays just to dereference a single 20 bit pointer. Considering that a "gate delay" in the brain is on the order of 10ms, this is a non-trivial matter. Of course, a neuron can do a lot more than a single CMOS gate.

Conclusions

The visual routine processor and the visual search engine built by adding a shift register to the VRP are *architectural* in the sense I described above: they are big chunks of mechanism that are shared across tasks and agents: The same VRP is used on both Bertrand and Ludwig. I also intend to port it to Elvis, our latest robot, and translate our visual navigation algorithms into VRP operations. I'd also like to port Ludwig's language system to Elvis. What makes the VRP an interesting architecture is not that it is "right" in some absolute sense, but that it demonstrates that we can do more with less than we would have previously suspected. Moreover, simplifications allow it to run very efficiently.

Ludwig's *language*, however, is utterly boring. Register-transfer languages are a pain to code in. I used it only to prove to myself that Ludwig really wasn't slipping any of Lisp's features in through the back door.

There's no reason why we all have to agree on a definition of "agent architecture" at this point. However, I do find the architecture/language distinction useful. As with KR languages and planners, it is tempting to introduce more and more features into an agent architecture until either it is too cumbersome to do anything, or the architecture is largely indistinguishable from a nondeterministic lisp. In either case, such an architecture tells us little about the essential nature of intelligence and embodied activity. Such a system is more a comment on how difficult it is to build intelligent agents than on how any particular set of features simplifies it.

References

[1] Philip Agre and Ian Horswill. Cultural support for improvisation. In *Tenth National Conference on Artificial Intelligence*, Cambridge, MA, 1992. American Assoiciation for Artificial Intelligence, MIT Press.

[2] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automoation*, 2(1):14–23, March 1986.

[3] Jonathan H. Connell. Extending the navigation metaphor to other domains. In Marc Slack and Erann Gat, editors, *Working notes of the AAAI Fall Symposium on Applications of Artificial Intelligence to Real-World Autonomous Mobile Robots*, pages 29–35, Cambridge, Massachusetts, 1992. American Association for Artificial Intelligence.

[4] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings, AAAI-92*, 1992.

[5] Ian Horswill. Nlp and real-time vision without pointers. Submitted to IJCAI-95.

[6] Ian Horswill. Visual routines and visual search. Submitted to IJCAI-95.

[7] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[8] D. E. Rumelhardt, J. L. McClelland, and the PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, 1986.

[9] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

[10] Terry Winograd. A procedural model of language understanding. In *Readings in Natural Language Processing*, pages 249–266. Morgan Kaufman, 1986.