Contents

Production systems

Jocelyn Paine

June 10, 1997

The PopBeast program is not suitable for you to experiment with, largely because it's based on old (hence free) software which is not robust.

I have therefore adopted a different approach for the agents that you will be experimenting with. This is based round the idea of a *production system*. After saying something about differing approaches to AI, I shall go on to describe how production systems work, and how they can be applied to Eden.

1 Introduction — some problems with classical AI

In the handout on *AI and PopBeast*, I described a simple artificially intelligent program. Its mental functioning is clearly separated into three parts. There is a perception box that takes "visual" input and builds from it a description of the world. There is a reasoning box that takes this description, together with a goal generated by analysing the last command, and builds a plan. And there is an action box that executes each action in the plan in turn. This three-way division is typical, as I've already said, of what is sometimes called "classical" AI.

I also mentioned that many AI workers no longer accept this separation as the best way to build robots that can act in an uncertain and changing world. There are many reasons why. Here is one, as an example: building a complete description of the world from sensory data is a very difficult job, and demands a lot of computer time. We can't yet do it for data coming from any but the crudest sensors. Even if we could, our description-builders could not possibly run fast enough to have finished computing the description by the time it is needed. The robot would run into a door before realising that it was there.

Moreover, it seems obvious (with hindsight) that the type of description you need depends on what you are trying to achieve. A robot typesetter, looking at these notes, would have to build a painstakingly detailed description of them in terms of sentences, paragraphs, tab settings, fonts, and margins. However, if I were to drop them out of my window, a robot street-cleaner need only classify them as "scrap paper" in order to do *its* job — that of throwing away anything that's not a person, dog, cat, or piece of street furniture.

As another example, when crossing the road, it's sufficient to perceive a speeding car as something to be avoided — quickly! A simple approach-time calculation based on how fast the image grows on your retina is sufficient to get you out of the way. On the other hand, more information may sometimes be useful. If you're a cyclist, it might be useful to recognise "BMW" as a special type of car, because it's typically driven by drunken agressive males and hence to be avoided from a long way away.

The paper *An emerging paradigm in robot architecture* by Malcolm, Smithers and Hallam (PSY AI box photocopy M149) is a nicely-written not-very-technical account of classical AI in robotics and of some newer alternatives. Most of it should be intelligible after reading the *AI and PopBeast* notes, and I recommend it.

Although PopBeast does not address these problems, it does have some educational value, in that it can demonstrate ideas such as multiple representations, logic-based world-models, path planning, and many other concepts which are themselves still widely used. In addition, it's an important demonstration of history, as well as of some current practice, such as the FMC self-guiding vehicle mentioned in the above paper. And those who don't know their history are condemned to repeat it.

However, I can't let you experiment with it because it isn't robust enough. Its planner is based on a classical planner called Warplan, written in the mid-70's. Although an important advance in its time, it can't solve problems much more complicated than the "open the door" one — on trying, it frequently gets into infinite loops. This is not surprising. It has been shown that this kind of planning is computationally intractable. Roughly speaking, the amount of time needed to make a complete plan increases with the exponential of the number of objects and plan steps being reasoned about.

One can circumvent this by giving the planner more knowledge about specific objects and tasks, by using clever heuristics, by trying to adapt previous solutions to similar problems, and by planning in less detail. However, although such programs do exist, even finding them takes a lot of work. I have therefore taken a different approach in this practical.

1.1 Reactive planning and nouvelle AI

PopBeast goes to one extreme — trying to build a complete plan of action before doing anything. Some people believe that it's time to experiment with the opposite approach — thinking ahead as little as possible, and reacting directly to the real world wherever possible. This is one of the differences between "classical AI" and what are sometimes known as the "nouvelle AI" or "artificial life" approaches. Using the term "nouvelle AI" may be a bit risky now: it was current in the early 90's. However, I'll do so in this section because I refer to some papers that do.

Trying to generalise over the subject is risky. But I think it's fair to say that the typical nouvelle AI approach is this: conventional AI has had such little success that we need something completely different. Most of what's needed in a natural intelligence was placed there by evolution long before the human brain evolved (compare the two billion years life has existed with the history of the human race). Conventional AI has concentrated on rational intelligence, printing the postage stamp on top of the Empire State Building without worrying about the building itself. What we should be doing is to work our way up the evolutionary tree, making a complete creature at each step and then building the next on top of the last. So we should start by trying to mimic insects. The result won't be able to solve chess problems or prove theorems, but it should be able to wander around the lab without getting stuck under chairs, running out of energy because it's failed to notice that its batteries are running low, falling over and being unable to right itself, or attacking its right forward wheel because it doesn't realise it's part of its own body.

Once we have a working insect, we can work up from there. Eventually, we shall find that in the course of building competance to operate in the real world, we have most of what's needed for rational thought. For example, spatial problem solving may fall out as a byproduct of visual navigation. The last few steps will then be easy.

The table below summarises the differences between an extreme classical AI position and an extreme nouvelle AI one. Most workers will not be so extreme, and will end up somewhere in between. Incidentally, there are many dimensions of variation between different AI techniques. The typical classical approach and typical nouvelle approach differ in several of these. This is not evident from the table below.

Classical AI	Nouvelle AI
Make a detailed plan in advance	React directly to the world
Store information about the world in memory	Take it directly from the world when needed
Decode perceptions into world-models	Use raw perceptual data whenever possible

1.2 The middle way

One way to resolve the conflict between these two extremes is to realise that cognition, whether it's classical planning as outlined above, or some other kind of computation, is only one way to achieve optimal behaviour. Other solutions are often possible, involving specialised perceptual systems or other "hardware" solutions instead of reasoning. David McFarland and Thomas Bösser demonstrate this for animal intelligence with a number of examples in their book *Intelligent Behaviour in Animals and Robots* (PSY). A more recent, and quite excellent, review, which has a lot more to say about the place of human cognition, is Andy Clark's *Being There* (PSY).

It is too early yet to say which approaches will give us the best robots. Taking a broader view, we might say that AI should study and compare all possible designs for intelligent agents, including those typical of both classical and nouvelle AI, pushing each to its limits in order to discover its strengths and weaknesses. A nice example of such a comparison, and a useful look at nouvelle AI from the viewpoint of someone who was once very classical, is in *Robot planning* by Drew McDermott (PSY AI box photocopy M165). That paper is a bit technical in places; one that's less so, and that looks at nouvelle AI from the point of view of a disillusioned classicist, is *Elephants don't play chess* by Rodney Brooks, in *Designing Autonomous Agents* edited by Pattie Maes (PSY KH:M 026).

2 Production systems

I have experimented with various AI techniques, but most of them were too time-consuming to build and too complex to demonstrate without a lot of pedagogical development. However, one simple one can be adapted to demonstrate various ideas from both the nouvelle and the classical AI approaches. This is the *production system*.

The idea of a production system is that you have two kinds of "memory" or storage. One — *short-term* or *working* memory — holds the data currently being worked on, as a set of symbols. The other — *long-term memory* — consists of rules which operate on these symbols, looking for certain patterns and replacing them by other patterns. You can think of them as stimulus-response rules: given a certain stimulus or pattern of symbols, your response is to replace it by a different pattern. Abstractly, this situation is shown below, in figures 1 and 2. The numbers, incidentally, are not part of the rules, and are just there for reference.

---

(1) if $\dagger\theta\flat$ then $\Box \smile \angle$
(2) if $\Box \sim \dagger$ then $\theta\Box \frown \Xi\dagger$
(3) if $\natural\Sigma\delta$ then $\heartsuit\Sigma\angle\delta$
(4) if $\Box \smile \angle$ then $\Box \sim \dagger$

Figure 1 Some production-system rules.

$\dagger\theta\flat$

Figure 2 Symbols in working memory.

I have not said anything about what these symbols might mean. The production system itself has no knowledge of this, and its operation does not depend at all on their meaning. The symbols only gain their meaning by virtue of their relationship to other symbols, rules, perceptions, and motor commands.

2.1 How a production system works

A production system works by repeating the same cycle over and over again. Each cycle consists of three stages: *recognise*; *resolve*; and *act*. In the first stage, the system searches for all the rules which *match* the symbols in working memory — that is, all the rules whose left-hand sides are somehow satisfied by these symbols. In the second, resolution, stage, it chooses one and only one of these rules. And in the third, act, stage, it *fires* it. This means that it adds the symbols in its right-hand side to working memory.

How would this work with the example shown above in figures 2 and 1? Our initial working memory contains the pattern $\dagger\theta\flat$. Looking at the left-hand sides of the rules, only one rule matches — number 1. So the first stage — recognise — returns just one rule. This makes the resolve stage trivial — it returns the same rule. Finally, the act stage fires the rule. This adds the symbols $\Box \smile \angle$ to working memory, which now looks like this:

$\Box \smile \angle$
$\dagger\theta\flat$

That's the end of the first cycle. We now go round again. This time, two rules match: number 1 and number 4. So the recognise stage returns these two rules. What is the resolve stage to do? There are various possible answers. One method, used in Eden, is to prefer rules that haven't been used recently over those that have. The system notes when each rule fired, and chooses the least recently fired to fire again.

The justification of this is that it prevents the system applying the same rule over and over again, and so helps to stop it getting stuck in a loop. In the jargon of production systems, we could call this resolution by inverse rule-firing-recency. It would sometimes also be called resolution by rule-refractoriness, the idea being that recently-fired rules are supressed by less recently-fired rules, and suffer a "refractory" or non-firing period.

In our example, rule number 4 is definitely less recent, since it hasn't been fired at all. The system therefore chooses it in preference to rule 1. The act stage can then add the symbols on the right-hand side of rule 4 to working memory, giving this:

$\Box \sim \dagger$
$\Box \smile \angle$
$\dagger\theta\flat$

We are now ready for the third cycle. This time, three rules match, rules 1, 4, and 2. Rule 2 is less recent than the others, since it has never fired. So the resolve stage chooses it. Firing it then adds $\theta\Box \frown \Xi\dagger$ to working memory, giving:

$\theta\Box \frown \Xi\dagger$
$\Box \sim \dagger$
$\Box \smile \angle$
$\dagger\theta\flat$

Now the fourth cycle. Again, only three rules match, the same as above. So the recognise stage returns them. What should the resolve stage do? If we go strictly by recency, then rule 1 is now the least recent, so we would choose it, and fire it, adding $\Box \smile \angle$ again. However, taking this approach isn't very helpful, because we will continually add the same pieces of data over and over again.

An alternative approach might be to say that once a rule has fired, it should never be allowed to fire again until it can add something different. This would eliminate all of rules 1, 2, and 4. And there are many other possible resolution methods. Which ones we choose depends on what we are using the system for.

2.2 So what are production systems for?

The simplest way to regard them is just as a programming language. Such a language contains just one construct — the rule — and so should be easy to learn; unlike (say) Pascal, with its range of data types, loop structures, conditionals, and other statement types. There have in fact been several production-system languages, most used from the mid-70's to the mid-80's. The best-known is OPS5, promoted for writing expert systems.

Although it's easy to understand how production systems work, it is not in fact easy to write big programs, and production-system languages have fallen out of favour. The problem is that it can be hard to see exactly what the resolution strategies will do, and hence in what order rules will fire and data be added to memory. I have added some features to the Eden production system which avoid some of these problems.

The main reason for being interested in them is their use as cognitive models. The notion of a production system probably dates from the 40's, when some logicians used them as idealised mathematical models of computation, in the same spirit as a Turing machine. As far as psychology is concerned, they became important in the early 70's, when Newell, Simon and Shaw adopted them as a model of human cognition. The idea was that, at the lowest level, the brain is of course built from neural hardware. However, this hardware implements short-term and long-term memories, rule matchers, and other components of a production system, in the same way that the logic gates in a computer implement adders, registers, and other components of a digital computer.

We can understand how a computer program works by explaining its operation in terms of adders, registers, and so on. It is not necessary to know how these are built from logic gates. In the same way (say the production-system theorists) we can explain cognitive processes in terms of the rules that are fired, the symbols that are added to STM, and so on. It is not necessary to know what is going on at the neural level.

2.3 The functional architecture of cognition

This view is often expressed by saying that the the neurophysiological hardware of the brain implements a "cognitive processor", a simple symbol-manipulating machine, and that this is a production system. For a statement of this view, see *Foundations of Cognitive Science* edited by Posner (Springer-Verlag 1989; PSY BH:P 084), Chapter 3.

But there can be many different types of production system. Here's a list of some of the ways they could differ:

1. The symbols allowed — what, and how many?
2. The resolution methods. You will see below that Eden has various other methods I haven't yet mentioned.
3. The capacity of STM. Is there a maximum size, and if so, what happens when more symbols are added? Do old ones drop out? If so, how does this depend on the time they were added? Is there any kind of "chunking" mechanism, so that symbols can be grouped into a single slot?
4. The rules' conditions. Can these test perceptual data directly, or must they go via STM? What is the maximum size of pattern you can have? Can you have "wild card" patterns — so that □_∠ for example, might match any set of symbols starting with □ and ending in ∠. Can the conditions check how many of some symbol there are?
5. The rule's actions. Can these generate actions directly, or must these go via STM? How many symbols can they add? Can they delete symbols?

Different theorists have proposed different types of cognitive processor. The ways in which one differs from another: the complexity of its rules, the capacity of STM and so on, these define its *functional architecture*. In the same way, a VAX, a PC, and an Apple Mac are all computers, with the same general design. But the details of their functional architecture — the maximum size of number that can be stored, the memory capacity, and so on — are all different.

For those who want to follow up such topics, there is a good account of how production systems work in pages 190–199 of the *Handbook of AI* volume I, by Barr and Feigenbaum (Kaufmann 1981; PSY KH:B 27). There are some famous production-system models of children's development: these are introduced in *Computers and Thought — a practical introduction to AI* by Sharples, Hogg, Hutchinson, Torrance and Young (MIT press, 1989): PSY KH:S531. The start of chapter 7 places production systems in a psychological context, while pages 226–231 in chapter 8 describe their use in modelling childrens' subtraction. This is also described in *Learning and Problem Solving 3*, Open University Course D303, Block 4, Units 26–28 (in the Oversize section, AA:O 2–O M), and in the Young and O'Shea article in *Cognitive Science* for 1981. Note: there is a misprint in *Computers and Thought* that sets this article in 1982.

Why did Newell and Simon consider production systems good theories? There's an early article by Newell, detailing some of the flaws he finds in experimental psychology, and what production systems have to offer. This appears in *You can't play 20 questions with nature and win*, in *Visual Information Processing*, edited by Chase (PSY BC:C 39).

Another view is given in Richard Young's chapter on *Production Systems for Modelling Human Cognition* in *Expert systems in the micro-electronic age* edited by Michie (E.U.P. 1980; PSY KH:M 58). One of the most recent approaches is Newell's SOAR, in *Unified Theories of Human Cognition* by (Harvard 1990; PSY BH:N 044). You will find a short survey of the book in *Behavioural and Brain Sciences* volume 15, number 3, September 1992. This also contains a number of criticisms of SOAR and the idea of unified theories.

3 Production systems and Eden

For some examples, I'll finish by looking at a simple example I used to use in the original Eden, the one described in the PopBeast handout. You won't quite be running this, but it's easy to describe, and easy to compare with the way PopBeast worked.

Let's start, by looking at a simple bug, one whose abilities are limited to grabbing and eating those pieces of food within its immediate range of vision.

So far, I've been pretty abstract about how production systems work. It's now time to get more specific. What sort of rules do Eden bugs use? What is their functional architecture of cognition? That is, what sort of conditions can a rule have? How big can short-term memory be? What resolution strategies are there? What can appear on the right-hand side of a rule?

3.1 `psbug1`

To start answering these questions, have a look at the code in figure 3, below:

```
resolve [p,s,r].

stm [].

holding(+)       => exec([use]).
over(+)          => exec([grab]).
can_see(+)       => move_towards(+).
```

Figure 3 The source of `psbug1`.

What is going on in the above code? It's in three parts. The first, `resolve [p,s,r]`, specifies the resolution strategies to be used. These letters stand for priority, specificity and recency respectively, and the methods are applied in this order. Priority is irrelevant here. It's possible, when writing rules, to give some of them higher priority than others. But I haven't done this. Specificity refers to the complexity of conditions. This doesn't matter here either, since all the conditions are equally complex.

So the resolve stage first looks for the rules with the highest priority. Since they all have the same priority, this doesn't help. It then looks for the rule with the most specific — most complex — condition. This doesn't help either. Finally it sorts the rules by recency, choosing the least recently fired.

The second part of the code, `stm []`, specifies the initial contents of short-term memory. The brackets with nothing between them mean that it's empty. It's actually going to stay empty throughout the life, since this bug reacts directly to its perceptions, and doesn't build up any memory structures.

The third part is the most important — the rules. Each rule is separated into two parts, a *condition* and an *action*. Before explaining these further, it's necessary to explain a bit about the bug's sense of "vision".

3.2 Bug's vision and the sensory predicates

You may remember from the PopBeast handout that the bugs see a 5 by 7 rectangle. This is oriented as shown in figure 4.

```
.  .  .  .  .
.  .  .  .  .
.  .  .  .  .
.  .  .  .  .
.  .  .  .  .
.  .  B  .  .
.  .  .  .  .
```

Figure 4 A bug's visual field.

In this diagram, the bug is facing upwards, or "north". It perceives this rectangle as a two-dimensional "retinal image". Each location in the retina represents one square, and contains a space if the square is empty, or a symbol such as + or * if it isn't. A bug brain can access this retina by various *sensory predicates*, such as `can_see` and `over`. These implement simple logical tests on the visual field. For example,

```
can_see(+)
```

is true if there is a + somewhere on the retina, and false otherwise. Similarly,

```
can_see('1')
```

would be true if the bug can see a `1`.

The predicate `over` is a more specific test than `can_see`. It is true only if the bug is in the same square as the specified object. Thus

```
over(+)
```

is true if the bug is in the same square as a piece of food. This is a useful test, because you can only grab things that are in the same square as you.

The predicate `holding` has nothing to do with the retina; it tests what the bug is holding. Thus

```
holding(+)
```

is true if the bug is holding a piece of food. It would be false if the bug is holding something else, or if its hand is empty.

3.3 Generating actions

You can see from the rules in figure 3 that the sensory predicates are used on the left-hand side, as conditions. What about the right-hand sides?

The simplest thing the bug can do is to issue a primitive action such as

```
exec( [use] )
```

or

```
exec( [forward] )
```

As well as the primitive actions, there are also some *motor competances*. These are little chunks of program which combine the primitive actions so as to do useful things. `psbug1` uses only one of these: it is

```
move_towards(+)
```

This causes the bug to move a bit nearer a piece of food. If it is facing it, it will move one square in the appropriate direction; otherwise it will turn towards it.

3.4 Putting these together — the rules

To recap,`psbug1` had the following three rules:

```
holding(+)       => exec([use]).
over(+)          => exec([grab]).
can_see(+)       => move_towards(+).
```

In English, these translate as follows:

```
If you are holding a piece of food, then use it. ``Use'' is
a general-purpose action: for food, it means eating.
If you are in the same square as a piece of food, then grab it.
If you can see a piece of food, then move towards it.
```

I hope it's now clear how these work.

3.5 Another brain — `psbug2`

Now let's look at behaviour in a more complex world. This one contains a coin (symbol `1`) and a slot machine (symbol `S`). The bug can pick up the coin and carry it to the slot machine. If it then puts it in, the coin will vanish and a piece of food will appear on some random square in the vicinity of the slot machine.

```
resolve [p,s,r].

stm [].

holding(+)       => exec([use]).
over(+)          => exec([grab]).
can_see(+)       => move_towards(+).

can_see('1')     => move_towards('1').
over('1')        => exec([grab]).

can_see('S'), holding('1') => move_towards('S').
next_to('S'), holding('1') => turn_towards('S').
facing('S'), holding('1')  => exec([use]).
```

Figure 5 The source of `psbug2`.

The top three rules are the same as those of `psbug1`, and deal with the case where there is food to be seen. But what if there isn't? In this world, it makes sense, if you can see a slot machine and you have a coin, to head towards the machine and use the coin. (For slot machines and coins, "use" puts the coin in the machine.)

If you don't have a coin, it's sensible to try acquiring one. The find-and-grab-coin behaviour is controlled by the fourth and fifth rules: I've put a blank line before them to make them easier to see as a separate group.

The find-and-use-slot-machine behaviour is controlled by the final three rules. Notice that, unlike the others, these each have *two* perceptual conditions on the left-hand side, combined by a comma, the symbol for "and". So the first rule means

```
If you can see a slot-machine, and you are holding a coin,
then move towards the machine.
```

There is an added complication here. You can only use a coin in a machine if you are facing towards it. Although in most cases, moving towards the machine will leave the bug facing the right way, this isn't necessarily so, e.g. if he backs towards it. Hence I have had to add a rule to make him do so, the one which activates `turn_towards('S')`. Note that I have also introduced two new perceptual conditions. These are `next_to`, which is true if the bug is standing on an adjacent square to an object; and `facing` which is true if he is next to an object and facing towards it.