Shifts in reasoning about software and hardware systems: do operational models underpin declarative ones?

(Early draft based on an invited talk given at the 1991 PPIG Workshop)

Shifts in reasoning about software and hardware systems: do operational models underpin declarative ones?

Marian Petre
Insitute of Educational Technology, Open University

Abstract:

Having studied expert programmers and hardware designers, I have long had difficulty with the notion that people program "solely declaratively", or even that they reason solely declaratively about systems, and I have been unable to find any expert who does. Expertise and operational models seem *at least* to be wholly coincident. People who begin with a "purely declarative" model of things soon find or develop for themselves an operational one, correct or not. I suggest that operational knowledge underpins our ability to reason declaratively—that a reasonably comprehensive operational model of the underlying 'machine', extending one or several layers deeper than the current focus, is necessary to our ability to design systems competently. The crux is time; what distinguishes functional from operational views is, primarily, time (also seen as process, change, behaviour). In a formal system, like a computing environment, there is a notionally perfect notational world sitting on top of a real world, and the mapping between them is not always perfect or ideal. The real world is imperfect, imposing physical and temporal constraints that matter to solution. Hence, we need to "escape from formalism"; we need knowledge of what lies underneath. The operational model is a way of coping with reality. The talk will address this question of how fundamental and how extensive is our need to know "how it works"; examples will be drawn from studies of programmers and of digital electronics engineers.

1. Introduction:

In the context of software and hardware systems design, how fundamental and how extensive is the need to know "how it works"? And, if we *do* need operational information, how deep must it be?

Having studied expert programmers and hardware designers, I have long had difficulty with the notion that people program "solely declaratively" or even that they reason solely declaratively about systems, and I have been unable to find anyone who does, although I have met many designers who include declarative-style reasoning in their bag of tricks.

Why am I concerned about declarative reasoning? Because of comparable developments in both software and hardware design:

In programming, there has been a move toward declarative—and I will emphasise functional—paradigms and toward the whole idea of offering abstraction building tools and being able to shift the complexity burden away from the attention of the designer.

Comparably, in hardware, there has been a shift, particularly in digital electronics, toward functional reasoning. The nature of digital electronic design is that, by reducing the electronics to noughts and ones, it allows the designers to forget about the electrical properties for the purposes of the design they're doing. It's a substantial abstraction. And, in fact, many of the developments in terms of the kinds of ICs that are on the market are about offering little black boxes that provide general-purpose functions. These are fairly complex little devices, but that have very *simple* descriptions. And as long as the designer can use that device by looking *only* at the description, the designer will buy it. So there's been a real shift toward building and reasoning with functions.

Despite these developments, which apparently evolve to cope with complexity, both disciplines retain alternative, operational reasoning styles. Are these historical fossils (of training, socialization, or evolution) or necessary antecedents to this functional development?

I suggest that operational knowledge *underpins* our ability to reason declaratively—that a reasonably comprehensive operational model of the underlying 'machine', extending one or several layers deeper than the current focus, is necessary to our ability to design systems competently. The argument will be illustrated and supported with examples drawn from all of my studies of hardware and software designers; unattributed quotations are from interview transcripts.

2. People use operational models:

There is evidence that people use operational models, and that they discuss systems, even declarative ones like mathematical proofs or PROLOG programs, operationally. I'll summarize a few examples.

2.1. Prolog evidence:

The PPIG community has a number of PROLOG researchers (e.g., du Boulay, Ormerod, Fung) who provide examples, but, in general, PROLOG is difficult to learn, at least in part because there are few cues at the language surface about 'how it works'. Giving novices some model of operation helps. If they're not given enough information, they borrow from other models they have (often erroneously) (Taylor, 1990).

2.2. Traces in code:

Throughout my studies of expert programmers, declarative programmers left traces in their code that showed that they had written operationally. The traces demonstrated that

they worked from the language implementation, not exclusively from the language definition. These would take advantage of some discrepancy between the language definition and the language implementation—exploiting some imperative "hacks" that were needed to make the language work, e.g., this Miranda example:

```
leapyear x = True,  x mod 400 = 0
           = False, x mod 100 = 0
           = True,  x mod 4 = 0
           = False, otherwise
```

In Miranda, although the order-of-evaluation of guards in a case structure is specifically "not defined" in the language, the implementation causes a textual order evaluation. Guards should properly be mutually exclusive, but in many Miranda programs they are progressive.

2.3. The declarative reading experiment:
Fundamental to declarative programming is the 'declarative reading'. But the notion that there is such a separate, accessible 'declarative reading' is, I think, suspect. In a brief study I did some time ago (Petre, 1989, Chapter 7), ten expert declarative programmers were asked to give an example of a declarative reading. They provided their own examples, most of them textbook examples of things like a Fibonacci numbers generator or least common denominator function. In that sense, they could be considered 'ideal' examples for extracting declarative readings.

None of them offered a pure declarative reading at their first attempt, and three of them *never* achieved a declarative reading, although they considered the operational reading they did give as declarative.

The not-strictly-declarative readings they gave were cast in operational language (e.g., "and then you do"; "after that"; "substitute"; "computing"), but this was not wholly explainable as a habit of language, because their readings also contained explicit references to evaluation mechanisms (e.g., iteration, unification, tail recursion), to the passage of time, and to behaviour.

2.4. Experts rely on operational information:
So, experts rely on operational information even within functional paradigms—even when they claim to be designing "wholly declaratively", when they're reading and writing declarative programs, when they're talking about mathematical proofs, and so on. (The reliance of declarative reasoning on operational competence was also suggested by Gunnar Moan in 1987.) Attention to declarative forms does not suppress concern for behaviour.

3. Does the non-expert also need operational information?

So, there is evidence that expert software/hardware designers *have* sophisticated operational models, no matter what paradigm they adopt. There is further evidence that

they *exert* these operational models even within functional paradigms. It is not clear if the evidence proves or even indicates that the operational model is necessary to expertise, or if expertise and operational models are just coincident. In any case, it's an interesting coincidence.

3.1. Non-experts develop their own operational models:
If we allow that operational models are somehow necessary, does the non-expert also need an operational model in order to design competently, or is the operational model something to do with the stuff that experts do and other people don't need to bother about? There is some evidence from the Prolog studies, and there is additional evidence from the way people learn electronics, that in fact everyone needs operational models to design competently, to produce competent solutions to anything but the very tiniest blocks. (And it may be that doing something that isolated doesn't constitute competence in the sense that I mean.)

Those who begin with a "purely declarative" model of things very soon find or develop for themselves an operational one, correct or not (examples available from both programming and electronics design).

> *"...in electronics, giving people a purely declarative model of things very soon leads to them finding themselves an operational one..."*

(e.g., The Conrad story—needing to know the lower level representations in order to find a solution. The simplified design rules approach.)

3.2. How extensive must the operational model be?
I suspect that any *competent* designer, at whatever level of expertise, needs an 'underpinning' of operational knowledge that extends below the level the designer is addressing, but not necessarily more than a couple of levels. The designer needs to understand and anticipate operational effects that will impinge on the solution at hand.

Consider an analogy from photography: The photographer doesn't worry how the individual transistor in the camera works, or how the little microprocessor in the camera actually executes instructions, or even how the software got written. What the photographer does look at is what sort of sensor the camera uses to measure light, because color response varies among sensors, that sort of thing. So, the photographer will look one or more two levels down into the operation of the camera, but not below. Similarly, the photographer will worry about the film in terms of response, light, time, ageing, and so on, but won't actually look into the dye chemistry or the manufacture of the film; a couple of levels down and stop there, before the photographer becomes a chemist or a film technician.

Furthermore, this operational model must keep pace with different tasks and with increasing expertise.

4. Why is the operational model necessary?

4.1. Complexity control:
So, functional or declarative approaches attempt to shift some of the burden of complexity away from the designer to the device, which increases in sophistication. In programming, the interpreter adds algorithmic information. In electronics, the integrated circuit becomes a building block, a complex object with a simple description.

The basis of digital electronics, and its crucial distinction from analog electronics, is that it facilitates functional reasoning by providing a set of rules which allows the designer to ignore the electrical properties and concentrate instead on the information content conveyed via signals.

> *"...we make it possible for the digital electronics designer to do work which is mainly digital and very little electronics...*
> *a functional view of digital electronics only starts to be possible when you can ignore the electronic half completely..."*

> *"...to do the best stuff you need to be able to think about it at the volts-amps-ohms level, and real signals running through real pieces of wire, about noise and other effects, about the physical results of putting electricity down a piece of wire...but if you thought at that level all the time, you'd never manage to build a computer...."*

One tradeoff is that the functional approaches control complexity in part by 'begging the question'; that is, they apply restrictions to what may be treated, disallowing or disregarding certain sorts of complexity, and making it difficult to address some sorts of problems (e.g., time and performance).

Another tradeoff is that, although this makes some kinds of designing easier or faster, we may lose invention or flexibility 'at the edges', and designs may be over-specified or unnecessarily complex. And so, in order to do competent real-world designs, one ends up having to address not only the functional level but also the operational things that underpin it.

4.2. Compensating for simplification:
Hence, logical or functional reasoning is a form of simplification, and the operational model is required when the simplification is inadequate.

Simplified representations, like functional programs or state machine descriptions in electronics, admit different implementations. The interpreter adds information. These different implementations, although correct with respect to the simplified representation, may differ subtly in ways that matter in reality—with consequent failure of a design, discrepancies of behaviour, or differences in performance. *"Subtle operational differences matter early to the behaviour of a system."*

Henry Lieberman of MIT agrees with this point of view. He told me a typical story about Prolog:

> *"...you pour true facts about the universe into the computer and then your program runs. So the typical paradigm to the declarative thing is that you tell it that the factorial of n is equal to factorial n times factorial of n–1, that's a true fact about the universe. But it's* also *a true fact about the universe that factorial of n is factorial of n plus 1 divided by n. If you give it that true fact about the universe instead, then Prolog goes off the deep end. You're left with how to explain which one of these true facts about the universe is the* better *true fact about the universe."*

Other examples of the intrusion of effects from outside the simplification are frequent in electronics, e.g.:

> *"...even in the toy or pedagogical sense, the gotchas...will come and bite you, and it's something that is taught quite early on in a logic course, because it's very hard to find with the debugging tools available...the problem is that the difference is subtle, and if you're teaching electronics at a first or second year level, and a circuit looks to a student like it must work because he's not seeing something that an expert would immediately see which is the effect of choosing the wrong sort...by subtle I mean the difference in the signals out of the counter, say, is very subtle, you can't normally see it with average pieces of test gear in a university lab, but it's sufficient to produce radically different behaviour in circuitry following it..."*

Such 'gotchas' matter in almost all real design, from very simple problems. And the reliance is on genuine competence, not just the right support: *"...an idiot with a great support system will still produce crappy software..."*

4.3. <u>The crux is time</u>:

The crux is time. What distinguishes functional from operational views is, primarily, time. From Steve Draper:

> *"...time is a device for not having everything happen at once...clocks stop all things from having all possible values at all possible times and getting into a terrible mess of arbitrary complexity..."*

The program or schematic design is static, but the system is dynamic; it changes. Electronics design describes a static object; the electronics themselves do not change over time. The state contained in the storage elements in that static piece of electronics changes, but the piece of electronics itself is inert. Similarly, a computer program is actually a static object, as witnessed by the fact that people print them out on paper. Again, a program will refer to things that have states, but is itself static. (Ignore for the moment the evolving state of a design as you're working on it, when it does change...). The operational instruction may refer to something which is an action at some time in the future, but that line of code is non-changing, and so an operational program is not complete without an idea of the machine which at some later date will have state. When people talk about what a program *does*, what they mean is 'what does a machine do when instructed by this program'. They're talking about programs *and* the device that

interprets them. A functional program carries the same query about how the underlying machine 'interprets'.

Declarative reasoning is essentially a static reasoning system (or a formalism for reasoning about static artefacts) does not help cope with change, with behaviour, does not afford a model of process. It restricts the 'universe' in this respect—although, in this restricted universe, it will solve your problems for you.

One of the critical issues in the real world is time, and there are *both* advantages and penalties to disregarding time for the purposes of design.

4.4. Infringements by the imperfect world:

One way to look at why operational models must underpin declarative or functional ones is the world is imperfect, imposing physical and temporal constraints that matter to solution.

> *"...in order to find where difficulty lies, you have to know something about how the system works...moral: if the systems are perfect, then having engineers who know nothing about the lower levels will work. Unfortunately, as soon as the system fails, the engineer is lost..."*

These 'imperfections' are also known as "implementation details" (details that matter in reality):

- performance
- predictability of behaviour (why admitting various
- implementations may cause trouble)
- physical constraints, e.g., memory size
- side-effects

So, the operational model is required for (at least) predicting behaviour, optimizing performance, debugging, and so on.

Correct-by-construction approaches exclude many of these imperfections, thereby reducing some sorts of complexity—but they take cannot take advantage of the sorts of problem reduction brought *by* those real-world constraints. The declarative solution must accommodate all possible values at all possible times; because declarative reasoning *does* not take time into account, it *can*not take time into account. It cannot take into account that a designer can just throw away a certain amount of complexity because it is precluded by the physical properties of the objects:

> *"...the correct-by-construction system will not take advantage of knowing that some inputs just weren't going to arrive"*

Consider baking a cake: the correct-by-construction system requires you to have all the ingredients before you start. The operational approach allows you to start baking before you get the cherry for the top, if you know the cherry will arrive later.

4.5. Escape from formalism:

These intrusions of time and imperfection demonstrate the importance of providing an "escape from formalism". There is a notionally perfect notational world sitting on top of a real world, and the mapping between them is not always perfect or ideal. Hence, we need knowledge of what lies underneath. The operational model is a way of coping with reality. In electronics schematics, this escape from formalism takes the form of textual annotations or addenda to the diagram—hints that rely on the designer's operational model. Functional programming rarely affords this escape. Essentially, this is a level-bridging exercise—between a useful notional simplicity/perfection/regularity, and reality.

5. An aside: Is there a relationship between functional reasoning and graphical representation?

Does functional reasoning lead to graphical representation or vice versa?

One of the things that seems comparable in the evolution of electronics reasoning and program reasoning is that at about the time there is a push toward functional paradigms, there is also a push toward graphical representation. Is there a relationship between these two, whether or not it is exclusive? Does functional reasoning lend itself to graphical reasoning? An informal survey of cases in which one sees graphical or textual notation suggests a proportional bias toward that association. A central issue is about making relationships clear. One of the things that graphics does well is make explicit connections between things, and relationships between things is the focus of functional reasoning.

So many of our 'notations' are largely declarative: architectural drawings (which don't mention flow of bodies through the building, etc.); engineering drawings (steam engine example); electronics schematics. They include only selected annotations regarding behaviour, detailed physical or operational descriptions. Consider also the evolution of schematic use in digital electronics:

- - the separation of functional layout from physical layout, and
- - providing different functional views of a given component.

6. Conclusion: bridging

An important advantage of higher-level reasoning and abstraction is the facilitation of mapping across domains. Appropriate simplifications may make it easier to see connections. Again digital electronics offers an example. The difference between digital and analogue electronics is that digital signals are ones and noughts, whereas analogue signals are variations in level. 2 + 2 in digital electronics yields 4. 2 + 2 in analogue gives 4 plus or minus a little bit. This is significant, because it means that addition in digital electronics is commutative, as it is in Mathematics. That's not necessarily true for analogue electronics. So, when the electronics designer uses the schematic representation for "add", it means the same thing that the mathematician means when he says "add", and the computer understands the same thing when it runs it. There is a mapping across these

three domains that means that this function, this notion of addition, is understood in the same way by all three.

> *"...by the wonders of digital electronics, the 'add' funciton that you put on the schematic behaves like the mathematician's 'add' and like the 'add' instruction in the computer... so that all three of you can be talking about the same thing ... the similarity of what goes on in the three domains is very close... so in general if you can prove that something works when you do it in mathematics, when you put the program together it works as well ... so what we have in the design of digital filters is an interconnection of functions to realize some mathematically-based design ... that interconnection of functions is then changed by a compiler to make a program which, when executed by a computer, runs ..."*

7. Contributing Literature

Bellamy, R. (1990) The use of pseudo-code in programming practice. IBM T.J. Watson Research Center Report.

Bonar, J., and Soloway, E. (1985) Preprogramming knowledge: a major source of misconceptions in novice programmers. Human-Computing Interaction, 1 (2), 133-161. DOI: 10.1207/s15327051hci0102_3

Du, Boulay, B., O'Shea, T. and Monk, J. (1981) The glass box inside the black box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14, 237–249.

Brooks, R.E. (1983) Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies,* 18 (6), 543-554.

Brooks, R.E. (1990) Categories of programming knowledge and their application. *International Journal of Man-Machine Studies,* 33 (3), 241-246.

Clark, A. (1987) Being there: why implementation matters to cognitive science. *Artificial Intelligence Review* (1), 231-244.

Collins, A. and Gentner, D. (1982) Constructing runnable mental models. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, Michigan.

Coombs, M.J., Gibson, R., and Alty, J.L. (1982) Learning a first computer language: strategies for making sense. *International Journal of Man-Machine Studies*, 16, 449-486.

Eisenstadt, M., Breuker, J., Evertsz, R. (1984) A cognitive account of 'natural' looping constructs. In: Shackel, Brian (ed.) INTERACT 84 - 1st IFIP International Conference on Human-Computer Interaction September 4-7, 1984, London, UK, 455-459.

Falzon, Pierre (1984) The analysis and understanding of an operative language. In: Shackel, Brian (ed.) INTERACT 84 - 1st IFIP International Conference on Human-Computer Interaction September 4-7, 1984, London, UK, 437-441.

Gallotti, K.M., and Ganong, W.F. (1985) What non-programmers know about programming: natural language procedure specification. *International Journal of Man-Machine Studies*, 22, 1-10.

Gentner & Stevens (1983) *Mental Models*. Taylor & Francis Group.

Guindon, R., Krasner, H. and Curtis, B. (1987) Cognitive processes in software design: activities in early, upstream design. In: Bullinger, H.-J., and Shackel, B. (eds.) INTERACT 87 - 2nd IFIP International Conference on Human-Computer Interaction September 1-4, 1987, Stuttgart, Germany, 383-388.

"The designer mentally explores and refines the mental model of the problem environment, especially through mental simulations."

Hoc, J.-M. (1977) Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies*, 9, 87-105.

Hook, K., Taylor, J., and du Boulay, B. (1988) Redo "try once and pass": the influence of complexity and graphical notation on novices' understanding of Prolog. Cognitive Science Research Paper No. 112, School of Cognitive Science, University of Sussex.

Jones, A. (1984) How novices learn to program. In: Shackel, Brian (ed.) INTERACT 84 - 1st IFIP International Conference on Human-Computer Interaction September 4-7, 1984, London, UK, 777-783.

Jones, A. (1993). Conceptual models of programming environments: how learners use the glass box. *Instructional Science*, 21(6) pp. 473–500.

Kowalski, R. (1979) Algorithm = Logic + Control. *Communications of the ACM*, 22 (7), 424-436.

MacLennan, B.J. (1982) Values and objects in programming languages. *SIGPLAN Notices*, 17 (12) 70-78.

Miller, J.R., Hill, W.C., McKendree, J., Masson, M.E.J., Blumenthal, B., Terveen, L, and Zaback, J. (1987) The role of the system image in intelligent user assistance. In: Bullinger, H.-J., and Shackel, B. (eds.) INTERACT 87 - 2nd IFIP International Conference on Human-Computer Interaction September 1-4, 1987, Stuttgart, Germany, 885-890.

Moan, G. (1987) PROLOGraph: a proposed system giving a graphical representation of a PROLOG execution. *(Manuscript).*

Ormerod, T.C., Manktelow, K.I., Robson, E.H., and Steward, A.P. (1986) Content and representation effects with reasoning tasks in PROLOG form. *Behaviour and Information Technology*, 5 (2), 157-168.

Petre, M. (1989) Finding a Basis for Matching Programming Languages to Programming Tasks. PhD Dissertation, University College London.

Sein, M.K., Bostrom, R.P., and Olfman, L. (1987) Conceptual models in training novice users. In: Bullinger, H.-J., and Shackel, B. (eds.) INTERACT 87 - 2nd IFIP International Conference on Human-Computer Interaction September 1-4, 1987, Stuttgart, Germany, 861-867.

Taylor, J. (1990) Analysing novices analysing Prolog: what stories do novices tell themselves about Prolog? *Instructional Science*, **19**, 283-309.

Winograd, T. (1975) Frame representations and the declarative/procedural controversy. In: G. Bobrow and A. Collins (eds.), Representation and Understanding. Academic Press. 185-210.

Young, R.M. (1983) Surrogates and mappings: two kinds of conceptual models for interactive devices. In Gentner, D., and Stevens, A.L. (eds.) *Mental Models*. Lawrence Erlbaum Associates. 35-52.
"Although it is widely accepted that people's ability to use an interactive device depends in part on their having access to some part of a mental model the notion of the "user's conceptual model" (UCM) remains a hazy one, and there are probably as many different ideas about what it might be as there are people writing about it. The common ground covers something like a more or less definite representation or metaphor that a user adopts to guide his actions, and help him interpret the device's behaviour." (p. 35)

Appendix: Quotes from preceding speakers at 1991 PPIG workshop

Steve Draper: "The operational model essentially is about time."

Henry Lieberman: "The problem of visual programming is to display dynamic motion through static structures." and "Programming is really a visualization process...you visualize in your head what the program is doing."

Paola Kathuria: "Declarative languages lend themselves to graphical representation."

Ben du Boulay: "In debugging you can't predict ahead of time where you need the detail."

Maurice Naftalin: "A program consists of (at least) two different kinds of information—logical and operational—so intimately linked that you can't fully understand either one in isolation." and "The right heuristics are exactly the operational intuition...provided we immediately verify them using the logical intuition."