On the difficulty of really considering a radical novelty

Derek Partridge
Department of Computer Science
University of Exeter
Exeter EX4 4PT, UK
derek@dcs.exeter.ac.uk

February 21, 1995

By means of metaphors and analogies, we try to link the new to the old, the novel to the familiar ... in the case of a sharp discontinuity, however, the method breaks down ... our past experience is no longer relevant; the analogies become to shallow; and, the metaphors become more misleading than illuminating. This is the situation that is characteristic of the 'radical' novelty. Dijkstra, 1989, p. 1398

Abstract

The fundamental assumptions in Dijkstra's influential article on computing science teaching are challenged. Dijkstra's paper presents the radical novelties of computing, and the consequent problems that we must tackle through a formal, logic-based approach to program derivation. Dijkstra's main premise is that the algorithmic programming paradigm is the only one, in fact, the only possible one. It is argued that there is at least one other, the network-programming paradigm, which itself is a radical novelty with respect to the implementation of problems on computers. And, as one might expect of a radical alternative, it shows much of the conventional wisdom concerning computing science, which Dijkstra variously attacks and dispenses, to be special pleading; not universals of computing at all. Finally, we explore what is known of this new paradigm in order to see what light it sheds on the fundamental problems that computing really does present to us. Not surprisingly, some become less problematic, some disappear altogether and others take their place, but, in general, it must be of benefit to modern computer technology to gain a wider perspective on the possibilities instead of seeing everything through the traditional tunnel of programming as formula derivation, and computers as the requisite, special type of symbol manipulation device.

1 Introduction

As he has done a number of times previously, Dijkstra has published an attack on the lamentable state of much of the practice in computing science (Dijkstra, 1989). His forceful argument for abandoning some of our cherished approaches to the subject and adopting a more demanding strategy offers much to take issue with as well as much to applaud, and a number of commentators have done both (see, for example, the seven responses appended to the original article). This paper is a contribution to the debate, but one that is totally different from its predecessors: for they all accept the nub of Dijkstra's argument but variously choose to challenge either its supposed unavoidable significance, or the remedies that Dijkstra prescribes. This article does not challenge his basic conclusions as such, nor the emphasis, and thus the relative importance, which he attaches to his claims. It challenges the assumption upon which he builds everything, as well as contesting the claimed universal validity of the crucial supporting assumptions. This challenge, if upheld, removes the universality, the claim to indisputable correctness, that Dijkstra attaches to most of his argument.

On the face of it, it seems unlikely that a pre-eminent computer scientist, and a group of similarly knowledgeable colleagues would all fail to see the essential limitation built into the fundamental assumption. Perhaps it has happened because the blinkers of logic and the dazzle of the light of absolute proof make it difficult to see other, radically different alternatives. And perhaps Dijkstra's warnings about human intransigence in according recognition to radical novelties has been a contributing factor. For as he says, "Coming to grips with a radical novelty amounts to creating and learning a new foreign language that *cannot* be translated into one's own mother tongue ... Needless to say, adjusting to radical novelties is not a very popular activity for it requires hard work. For the same reason, the radical novelties, themselves, are unwelcome" (p. 1398, author's emphasis). Whatever the reasons, a radical novelty that relegates all of Dijkstra's argument to a league of lesser importance has been missed or peremptorily dismissed.

What is being claimed here? Insert 'algorithmic' as a qualification to most of Dijkstra's references to computer programming and to computing science, and you will have scarcely interferred with the tough argument that he intended, yet you will have seriously blunted the cutting edge of his conclusions. But, most importantly, you will have made room for an alternative, non-algorithmic computing paradigm – in a phrase: a radical novelty. Furthermore, this newly cleared space is not necessary simply as a philosophical nicety, in effect an admission that we may not yet be seeing all possibilities (although even this is denied by Dijkstra whose article is peppered with absolutes that slam the door on the possibility of equally valid, alternatives). The fresh ground is, in fact, already staked out by at least one novel computing paradigm: there is, without doubt, a non-algorithmic approach to computing – network programming. It doesn't seem to suffer from Dijkstra's two radical novelties, and it doesn't seem to be in need of the harsh medicine that Dijkstra prescribes with such conviction. This is not to claim that network programming is trouble-free, far from it, it is merely to claim that it makes an irrefutable demand for a place in computing science at a time when Dijkstra (and many other computer scientists) would claim that the basic framework is already filled out, and what remains is to sort out the fine details – e.g., develop the right sort of predicate-logic language, or standardize the object-oriented approach (the specific personal goals depend upon individual biases). In classical Kuhnian terminology: normal science is holding sway in the various niches of the establishment while an incipient revolution is gathering credibility on the fringes.

And as this credibility grows to the point where this fringe activity can no longer be comfortably overlooked, the eventual acceptance of a radical, new paradigm for computing will introduce phenomena that the basic language of computing is strained to accommodate, so strained that the normal terminology is inadequate or simply wrong – e.g. 'programming' as the process by which an implementation is constructed. When confronted with a radical novelty, we are, as Dijkstra says, forced to create and learn a new foreign language.

First, let us look at Dijkstra's universals of computing, for they lay out the ground plan for computer scientists, who, though they may quibble with details, seem surprisingly in agreement that Dijkstra has got it broadly right. The nature of network programming tells us that this is not so.

2 Dijkstra's two radical novelties

"The underlying assumption of this talk is that computers represent a radical novelty in our history." Such is Dijkstra's opening sentence. It will be argued that he is right about this, but wrong about what the basic radical novelty really is. He contends "that automatic computers represent a radical novelty, and that only by identifying them as such can we identify all the nonsense, the misconception, and the mythology that surround them.

Closer inspection will reveal that it is even worse, *viz.*, that automatic computers embody not only one radical novelty but two of them.” (p. 1399).

The first radical novelty derives from the enormous span of the technology: “From a bit to a few hundred megabytes, from a microsecond to to half an hour of computing, it confronts us with the completely baffling ratio of 10^9!” (p. 1400). The radical novelty then is that “The programmer is in the unique position that his is the only discipline and profession in which such a gigantic ratio, which totally baffles our imagination, has to be bridged by a single technology. He has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.”

The second radical novelty derives from the nature of computers, the way we’ve chosen to build these artefacts, apparently. It “is that the automatic computer is our first large-scale digital device ... it has, unavoidably, the uncomfortable property that the smallest possible perturbations – i.e., changes of a single bit – can have the most drastic consequences.” (both quotes p. 1400).

We must now place these problematic characteristics in their proper context, that of algorithmic computing, or, as Dijkstra would have it, simply “computing.”

3 The nature of computers and programs

“Well, when all is said and done, the only thing computers can do for us is to manipulate symbols ... this is a discrete world and, moreover, ... both the number of symbols involved and the amount of manipulation performed is many orders of magnitude larger than we can envisage. They totally baffle our imagination, and we must therefore not try to imagine them" (p. 1401). He then turns his attention to programs and gives us: "The program is an abstract symbol manipulator which can be turned into a concrete one by supplying a computer to it ... [programs are] rather elaborate formulae from some formal system" (p. 1401). This takes us nicely onto the programmer's task: "He has to derive that formula; he has to derive that program. We know of only one reliable way of doing that, *viz.*, by means of symbol manipulation" (p. 1401).

"Hence computing science is – and will always be – concerned with the interplay between mechanized and human symbol manipulation usually referred to as 'computing' and 'programming,' respectively" (p. 1401).

Having set out the fundamentals that Dijkstra builds on, it is time to examine a new radical novelty of computing and see how these 'axioms' of computing science appear from the new viewpoint.

4 Network programming, a more radical novelty

From the earlier quote we can see the importance that Dijkstra attaches to the identification of the radical novelties: it is through them that we can correctly identify the nonsense and the misconceptions – in sum, the mythology of computing which has no part to play in the proper development of an appropriate science. Dijkstra claims to look closer in order to perceive his two radical novelties; he might have been better advised to step back to where, with the bigger picture in view, he might have seen that programming as conscious symbol manipulation, and computing as its automatic analogue are just one way to compute – surely the most important way, but certainly not the **only** way.

The computational subfield named 'neural computing' (or NC), 'parallel distributed processing' (hence, more manageably, PDP), or 'connectionism' encompasses a rich variety of approaches to computation. Such networks are typically composed of simple processing elements connected by weighted links which pass the input value received from a unit at one end, modified by the weight, to the unit at the other end. And if we restrict our consideration to just those networks in which collections of processing units are organized in layers from input layer through a sequence of hidden layers to output layer, then we have the MultiLayer Perceptrons (MLPs). MLPs can be trained (on a set of input-output pairs using the backpropagation algorithm) to implement a function, and when so trained such a network implements the function in a radically novel way in comparison to a logic-based formula type of implementation – i.e. a conventional program. I'll take this particular NC model as my exemplar of network programming (it is, in fact, the most commonly used model, by far).

Computation with such a network involves coding the input into the input layer from whence it becomes an array of activity values which are accumulated, thresholded (or 'squashed') and passed on from one layer to the next, in parallel, until the output layer finally exhibits some resultant pattern of activity which is then decoded into the output of the computation on the given input. It is usual, and significant for the novelty claim, for the activity values to be passed in parallel from one layer to the next. A computation is then a wave of processing through the network rather than a single thread of control through a sequence of individual logical expressions.

The implementation of the function is distributed (more or less) uniformly across layers of processing units in a network program, not dispersed as a set of alternative threads each along a specific control path. An enlightening metaphor here is of a computed result as a weight hanging from a computation that must (if it is to be correct) bear this weight without breaking.

A conventional program is a single, unbreakable steel wire. The guarantees of logic and proof ensure unbreakability; if you know that one wire is enough, then the main task is simply to concentrate on getting that one strand strong enough. The result is a clean and elegant, minimal structure, but one that we can put absolute trust in.

A network program, on the other hand, is a web of fine silk threads. Any one of which will easily break, but in total they hold the weight quite comfortably, so much so, that the loss of any relatively small number of these threads has virtually no effect on the overall strength of the web.

Very significantly, in the light of Dijkstra's second radical novelty, the output of a network computation is a continuous value that needs to be thresholded in order to yield a discrete result, and it tends to vary smoothly as network structure is varied. An accurate result, say 0.9999 which should be 1, will tend to degrade gracefully as the implementation is subjected to successively greater random destruction. The discontinuities which, as Dijkstra points out, are an unavoidable feature of algorithmic computing are largely absent in network computing. The undoubtedly discrete computation (on a digital machine) that underlies the network computation is so fine grained with respect to the overall computation that the perceived behaviour is, in effect, continuous – just as all continuous systems are founded on discrete atomic and molecular mechanisms. And, although it may seem like particularly bad news that the discrete computational elements are more primitive than those in a conventional symbol-structure program (extending the span of the technology), we shall see that in network programming conceptual management of discrete computational primitives is no longer an essential requirement of the implementation methodology. The programmer doesn't actively and directly derive a formula, rather the computer does it, in effect, with the aid of the training set. Similarly, the other activities that typically press the conventional programmer to consider operational detail – i.e. debugging and functional enhancement – become manifest in network programming as more-training and retraining. To what extent this practice can be successfully pursued without recourse to operational analysis is yet another open question, but the answer is not obviously the conventional one.

This is not to say that the conceptual opacity of network implementations is not a problem – it is, or it is perceived to be. There is currently much research on how to extract conceptually tractable, i.e. symbolic-formula representations, from network implementations. And there is progress towards symbolic-expression interpretation of network implementations, but it is both partial and approximate. It is, nevertheless, useful when reasoning about such implementations, but it is not even the beginnings of a symbolic interpretation such as conventional computing science insists on – a symbolic interpretation that must be both complete and precise (as well as satisfy a few more criteria, see later). Anything less than this is no better than useless; it is not even a small step in the right direction, from the conventional viewpoint.

However, there is more than a faint possibility that this urge to achieve conceptual mastery over the operational fine detail is more a hangover from algorithmic computing (bad conventional practice, as Dijkstra claims) than an important necessity to properly support network programming as a discipline for practical software construction. The network programmers may be unduly influenced by the language and practice of algorithmic computing, and have failed to learn all of the new language, for some of this 'learning' really means 'discovery.' A statistical justification for network reliability, which does not require operational detail, may replace a logical-inference based one which does need to be founded on the details of program infrastructure.

In the network-programming paradigm, the programmer's conscious design effort goes into initial network configuration – how to encode the input and decode the output, how many hidden layers of processing units, and how many processing units in each layer. Notice that it does not involve fine detail such as which processing unit to connect to which, and what initial weight to give to each link. Each unit in every layer (except the output layer) is connected to all units in the next layer, and initial weights are assigned randomly within some range of values.

So network design (although ill-understood and thus a somewhat black art at the moment) does not require that the programmer conceptually manages the fine detail that conventional programming demands. The network programmer designs a network that *is capable of* implementing the desired function, rather than one which instantiates it from the outset.

The deep conceptual analysis required of the network programmer is in terms of extracting a representative training set of examples (input-output pairs) from the problem to be implemented. A representative training set, that is, a set that will cause the network to converge on the desired (or an adequate) generalization, is yet another ill-understood notion.

Achievement of the desired generalization is measured by testing, just as for conventional software – which would appall Dijkstra. The main point, however, is not that network programming has all the answers for the future of computing, but much more modestly that network programming is a radical alternative which can be made to work even in our current state ignorance and lack of formal analysis of what's actually going on.

The undoubted fact that, at the moment, network implementations compute with an uncertain probability an otherwise ill-defined function (not very different from much conventional software, incidently) is not a reason to deny network programming admission to the fold of computing paradigms. It may be reason to be cautious about what we might usefully compute with this paradigm, but that is an entirely different concern. There is much work to be done on network programming, but it can only improve the situation to eventually yield a robust and reliable, radically new, programming paradigm, albeit with restricted application.

The implementation of a function as a network program (i.e., "programming" in algorithmic computing) is not a process of formula derivation, nor one of successive divide and conquer that yields the conceptually deep hierarchies with which the algorithmic programmer must grapple. And the product of the programming effort is not a formula, an explicit symbol structure at a useful conceptual level. A trained network is, however, a fixed and deterministic implementation of a function; it's just a different sort of implementation which is generated in a novel way. We can forget artificial intelligence, and any supposed brain-modelling possibilities; it is simply a radically new way to implement functions.

There are many ways to realize most of the details of network programs, and most (perhaps all) are ill-understood. There is, as yet, no science of network programming, but this is not the point. The important point for our current purposes is that there is a viable paradigm, named network programming; it is radically new with respect to much of the conventional wisdom in computing; and it throws new light on many of the supposed fundamentals of computing science.

5 Two radical novelties normalized

Computers really are large-scale digital devices whether we want to admit it or not; they are designed and built to be exactly this. This sort of statement seems to leave Dijkstra'a second radical novelty, at least, in an unassailable position, provided we are prepared to concede that (as Dijkstra claims) all other machines are significantly less discrete and also much less complex.

Let's look closer. There are two prongs to deflate the radicalness of this claim: firstly, discreteness is relative and level dependent, not an absolute characteristic of any system, and secondly, we build our computers in order to fit the demands of conventional programming, we can, and sometimes do, build analogue computers. It's our choice that computers are digital devices, not an unavoidable consequence of a need to mechanically compute.

To expand in little on each of these points: Computers are no more symbol manipulators than human bodies, say, are. It's just that we have found it a convenient and powerful way to view the actions of electron shuffle and charge transfer which is what computers really do (above the quantum-mechanical level). Indeed, they've been designed to be viewed this way; it is not at all clear that the human body was designed to facilitate a discrete perspective. But, with a little effort, the human body may also be viewed as a similar type of symbol-manipulation device. We just need to find the right alphabet and language with which to interpret its actions at some convenient level. The main difficulty is to discover a satisfactory underlying symbol scheme (with computers their construction has been in response to the desired symbol scheme – a much more sensible way to do things).

We also have to face both level and complexity problems in our attempts to view the human body as a symbol-manipulation device. Nevertheless, the classical sciences, biochemistry for example, are beginning to provide us with symbol schemes that the human body can be seen as manipulating. Some (small) parts of the human-body machine are quite well understood as manipulators of chemical equations. And, in principle, there is no reason why it could not be so for everything that the human body does. Does the human body then become a symbol-manipulation device? Or a very large-scale discrete device?

Perhaps it doesn't *become* either of these things, for they are not intrinsic properties, they are simply views that we can and do choose to impose, for our own convenience, on systems in general. In sum, one view of computers does give us Dijkstra's second radical novelty, but only because we choose to have it that way, and we do this because programming is the derivation of a symbol structure that will need manipulating. So perhaps it is the fundamental nature of programming which determines that the most productive pespective on computers is as large-scale digital devices. Symbol manipulation, especially when the constituent symbol structures are constrained to constitute a formal language with all its required qualities (such as referential transparency), *imposes* the discrete-system view.

Does programming have to be formula derivation? The answer is "yes" if programming means constructing the sort of object that a digital device, viewed as this special kind of symbol manipulator, can interpret precisely. But it is not so obviously in the affirmative if we view programming more broadly as an economic and convenient way to achieve the goal of a specific active system, i.e., a system that computes particular things. To steal the bones of a sentence from Dijkstra: a program is what turns a general-purpose machine temporarily into a special-purpose machine. The human body is, of course, a special-purpose symbol manipulation device, the program is 'hardwired'. Evolution, it seems, did not opt for this very convenient separation of fixed general ability and flexible specialization which can be coupled and decoupled at will.

'Must computation be realized as a process of discrete symbol manipulation?' seems to be the next question, and not a foregone conclusion, fundamental to all computing. Clearly, this is but one possibility, and it is the one that gives us conventional computing science. Equally clearly, there are other ways to compute and hence to 'program'.

Network-programmed computers are manipulating symbols, but not at the same conceptual level as conventionally programmed computers. At this point the argument is coming close to that of Fodor and Pylyshyn (1988) who mount a sophisticated and tightly argued attack on PDP networks as theoretically significant models of cognition – rather than merely an awkward implementation device. The basis for their argument is the very fact that such networks cannot be equated, in any straightforward way, with representations of symbolic expressions "at the cognitive level" (as they put it); they happen to use formulae in the propositional calculus as examples. And this, of course, is support for the view that network programs are not symbol structures in the same way that conventional programs, Dijkstra's formulae, are.

The difference in type of symbol structure is crucial. For the computing scientists are not talking about *any* sort of symbol structure, and *any* type of symbol manipulation – in both cases they are referring to very specific notions. To echo Fodor's and Pylyshyn's argument: it is essential that the symbolic formulae that are programs in conventional computing, have a combinatorial syntax and semantics which is also systematic – i.e. a rich variety of new structures can be formed from a collection of elemental structures such that these new structures can be deterministically checked for syntactic correctness, and, if a structure is syntactically correct, then its meaning is also well-defined. The basis for these requirements may be called *classical compositionality*; the epithet "classical" is required in order to distinguish this basic property from the PDP version. The formal languages used for conventional programming have just these general properties (more or less), the symbol structures that are networks don't.

Van Gelder (1990) has (most committed connectionists would argue) extricated the PDP cognitive modellers from the trap of 'mere implementation detail' by pointing out that classical compositionality is not the only sort of compositionality. He distinguishes the classical variety as *concatenative compositionality* as distinct from *functional compositionality* which can occur in PDP implementations. Hence, PDP implementations can be compositional (as Fodor and Pylyshyn require) without being classical.

The classical form of compositionality, which is the form that Dijkstra would have us believe is a universal characteristic of all programs, means an internal formal structure of a certain kind within an expression. To use van Gelder's words "the abstract constituency relations among expression types find direct, concrete instantiation in the physical structure of the corresponding tokens", and he further points out that this kind of internal structure is typically called "*syntactic* structure"(p. 361). However, the concatenative mode of combination implied by this classical type of compositionality "is not the only way of implementing the combination of tokens in order to obtain expressions ... there is no inherent necessity that these [systematic] methods [of generating and decomposing compound expressions] preserve tokens of constituents in the expressions themselves; rather, all that is important is that the expressions exhibit a kind of *functional* compositionality" which leads to a definition of this non-classical notion of compositionality (p.361).

> Functional compositionality is obtained when there are general, effective, and reliable processes for
>
> **(a)** producing an expression given its constituents, and
>
> **(b)** decomposing the expression back into those constituents (p. 361).

The validity and significance of these ideas is hotly debated, mostly by the cognitive science and philosophy community. The foci of the debate are: firstly, the 'level' of this notion (is it at the level of inherent structure in cognitive models or is it just an implementation-level distinction?), and secondly, the practicality for the cognitive modeller (even if it is a proper 'cognitive level' phenomenon, is it practically advantageous, or even possible, to model in a functionally compositional way?).

Fortunately, for the purposes of the current argument questions of cognitive-level validity and practicality for cognitive modelling are irrelevant. All protagonists accept the view that there is a radically different way to compute. Their disputation centers on whether it is more than just implementation detail, and whether it is a possible route to a new variety of cognitive models. For us, mere implementation detail is quite sufficient.

It is demonstrably possible to construct programs that are **not** symbolic formulae, such programs do not exhibit the classical compositionality hitherto accepted as a characteristic of all programs, and therefore a universal of computing. The classical approach to symbol manipulation, and expression derivation, is not the only one possible.

It is desirable, although not strictly necessary, to address the practicality issue, and we shall do so, but notice that is not the same practicality issue that cognitive modellers grapple with. For the software engineer or computer scientist, the practicality question centres on whether there is an effective methodology for implementing problems in a functionally compositional way, and whether there is any practical advantage in so doing. The answers are both in the affirmative, but most probably only for certain sorts of problems. And while generally-applicable, effective and reliable processes to exploit functional compositionality might be a bonus for the network implementor, the question of the existence of such processes (either in principle, or in practice) is not germane to the point being made.

On a somewhat different tack, Schwarz(1992) "argue[s] that there exists an important difference between connectionist and orthodox computation" (p. 208). He bases his case on the point that "connectionist networks collapse a dichotomy that is fundamental to understanding abstract computation: the functional distinction between a (finite state) processing part and a (potentially infinite) memory" (p. 208).

Schwarz characterizes all conventional computation as performed by Processing-Memory Systems (PMS): "what makes a physical system a PMS for some function *f* is that

(i) it realizes the finite state processing part of some abstract computing machine for *f* and

(ii) if it computes only some restriction of *f*, the result of idealizing away from its memory constraints computes *f*"(p. 215).

An algorithmic computation can be decomposed into finite and infinite parts — processing and memory, respectively — a network computation cannot. This scope for decomposition in conventional computation provides the basis for the very useful distinction between algorithm and data representation — from finite processing part, and (potentially) infinite memory-part, respectively. And this distinction is the basis for viewing a conventional computation in terms of the manipulation of symbolic representations. Network computations, which are not amenable to the initial decomposition (not being PMSs), cannot be interpreted in these terms.

Schwarz uses his arguments to make the point that the Turing-equivalence of neural networks is deceptive because it masks a fundamental difference: the necessary finitude of any physical computation is a totally different constraint within conventional and network computation — and is far more restrictive within the latter. But for our purposes, Schwarz's argument is simply further evidence that neural-network computation is indeed a radically novel paradigm, and that one basic point of novelty is that neural computations cannot be interpreted as manipulations of symbolic representations.

To turn attention to Dijkstra's first radical novelty – the excessive span of conceptual control demanded of the conventional programmer – this again seems to be an artefact of the conventional approach, and not a universal of computing at all. From the earlier discussion of implementation derivation (I hesitate to call it 'programming') in the network paradigm, it should be clear that the use of automatic training regimes relieves the implementor of the necessity to manage fine operational detail. In fact the distributed nature of the implementation of the function in such networks, together with the parallelism of network functioning, makes conceptualization of operational detail extremely difficult (which could be a radical answer to Dijkstra's quest for a way to make programmers eschew reasoning at the operational level). So what could be, and may yet turn out to be, a big problem for network programmers may, alternatively, be avoidable (as its certainly has been in some applications so far) because of the radically different nature of the computing paradigm. Apart from mere wishful thinking, the distributed and fine-grained nature of network implementations (i.e. the very things that make operational detail cognitively impenetrable) does appear to readily support a statistical approach to behavioural analysis. It may thus be that we see statistics (and geometry, as it happens), providing a formal basis for network programming, and taking a place alongside logic which has hitherto been promoted as *the* underlying formalism for computing – again, a supposed universal may be toppled from its unique position.

A further feature of the network paradigm, which suggests that the first radical novelty will not be operative in network programming, is that the absence of conscious algorithm design allows the network implementor to avoid the deep conceptual hierarchies which result from the traditional divide-and-conquer approach to the effective conceptualization of complex problems. This approach is a necessary precursor, it seems, to effective derivation of a complex formula, but not to the construction of trainable networks. Network implementations tend to be more monolithic structures. The main constraint on this seems to be that training time grows excessively as the goal function becomes more complicated. So there is research on modular networks for this reason, as well as for reasons of software reuse, but not so much for the traditional reason of building a complex computation out of conceptually manageable modules.

And a last significant point against the claimed need to manage an excessive span, from a bit to a few hundred megabytes, is that network implementations are not "digital systems", and thus do not suffer from minor perturbations causing major disruptions. A grasp of all fine detail is not required because the inherent redundancy in these implementations means that the final outcome does not depend of every operational detail being correct – break a few strands in the web and who would notice any significant change in overall behaviour?

"In the discrete world of computing, there is no meaningful metric in which 'small' changes and 'small' effects go hand in hand, and there never will be." (p. 1400) As an absolute claim about computing, network programming suggests it is wrong. This is not too surprisingly, as absolute claims seldom withstand the ravages of time; the real surprise is that the clear refutation of this claim, in the form of a large number and wide variety of network programs, was already in existence (for those that were prepared to see them) long before this claim was made. But as a claim limited to algorithmic computing, it is more defensible[1].

The reflections of Dijkstra's two novelties in the network-programming paradigm exhibit a tight coupling: the lack of a conceptually manageable, precise, symbolic interpretation actually extends the span of the technology, but the nature of the new paradigm is such that the programmer and the formal tools do not need to descend to precise management of the lowest-level elements. The programmer designs the final implementation (algorithm?) only indirectly, and the continuity of behaviour displayed by functional distribution over fine-grained elements can make a coarse-grained statistical approach reliable.

6 Network programming not a radical novelty

A large number and wide variety of problems have been programmed (if that's the word) as networks – e.g. speech pronunciation, sonar analysis, syntactic analysis of natural language (Widrow *et al.*, 1994, list and reference hundreds of applications). But, further than this, some problems are more easily and more satisfactorily implemented as networks than as

[1]in EWD1129 of 30th April 1992, Dijkstra has explained the reasoning behind this quotation, and, in particular, his interpretation of the word "never"

conventional programs, i.e., as algorithms (e.g. human face recognition, in the form of the WISARD system of Aleksander et al., 1984). Typically, these have been pattern-recognition problems for which a precise specification is hard to provide, yet examples of correct behaviour are abundant. Yet 'what is a pattern recognition problem?' perhaps more of a chosen viewpoint than an absolute characterisation of problem type. The reverse of this observation is, of course, that for problems which only have a specification there is no observed behaviour with which to train a network! A conventional program must be constructed in order to generate the training set, at which point a network implementation is hardly needed, unless it offers useful characteristics that the symbolic formula lacks (e.g. speed of execution).

Suffice it to say, the bare fact of existence of a large number of network implementations makes it difficult to deny that a network-programming paradigm exists. So the defence of the conventional wisdom, as espoused by Dijkstra, must rely on a denial that the paradigm really is radical in the way that has been claimed earlier.

A number potential lines of attack immediately suggest themselves: networks are really discrete, symbol manipulation devices and so are, fundamentally, no different from what I've called conventional programs; the so-called distributed implementation of the function in networks is merely parallel processing with 'smaller' symbols and hence will fit comfortably into algorithmic parallel programming; if networks really don't admit the right sort of symbolic interpretation, with total precision, then we'll never be able to prove the correctness of a network; the constituent functions of a network (i.e. summation, thresholding, and transfer of activity) are all programmed, so networks are conventionally programmed;

the network paradigm will never really scale up, a monolithic network implementing, say, an operating system seems unlikely in the extreme.

There is some truth in all these claims. Network programming does bear some relationship to algorithmic programming. It is not, after all, from another planet. The decision about radical novelties is one of where to draw the lines – when does an extreme become sufficiently extreme to be considered something new? The human body, as well as everything else, is really discrete if we care to go down to the level of quantum theory. And although this may be true in some undeniable sense, it is typically not very productive to view most systems at this level of discretization.

When conventional programmers talk of symbol manipulation, they don't mean any sort of juggling with any sort of symbols, they are referring to a very specific type of manipulation of a special class of symbol structures – i.e. structures that are formulae in a formal language of a special type, and manipulation that yields the proper semantic interpretation of these formulae. Given these, quite tight, constraints on what sorts of symbol manipulation conventional computing must be, it is clear that network-programmed computations are not in this class of symbol manipulation at all. The level of discrete symbol manipulation in many implemented networks is so far below the conceptual level, as well as being awkwardly distributed, that the discrete-symbol-manipulation view of network implementations can be no more than an in-principle one, at best. And the whole point for the formula-deriving programmer is that this association can be made in practice, and be made easily. For the conventional programmer, the computer is a device that does what he could have done himself, but does it more quickly and more accurately. For the network programmer, the computer is not an extension of his conscious symbol manipulation abilities, it is providing a different sort of symbol manipulation ability; one that the human brain cannot consciously perform at the level of conceptually useful quantities.

The parallelism objection is answered similarly – it is viewed, conventionally, as the same sort of symbol manipulation, with the extra problem of having to manage more than one thread of formula manipulation at a time. We might also add that parallelism in conventional computing embodies many open problems; it is severely taxing conventional techniques developed for the management of sequential computation. If coarse-grained parallelism (i.e. just a few parallel processes – sometimes called 'mere' parallelism) with readily interpretable symbol structures is proving difficult to master with conventional techniques, what hope is there for the massive parallelism of networks manipulating conceptually-opaque symbols? On the positive side: if statistics does give us the desired formal framework for network programs, then what is learnt may well spill over to the benefit of work in conventional parallelism.

This point takes us on to the third objection. It is difficult to see how traditional program logic can get a grip on network implementations and deliver proof-based verification of performance (although by no means a forgone conclusion). So, it might be argued, network programming can make no serious case for a place in the *science* of computing. Such a bigoted view puts an unreasonable bound on what computing science embraces. No other science is so demanding, except perhaps mathematics; it is, of course, those who view computing science as an extension of mathematics who are likely to be predisposed to this restrictive viewpoint. From a more 'shop-floor' perspective, network implementations do seem appropriate for statistical treatment, and it is debatable whether anything better than a statistical assurance of reliability is really attainable, in practice, for any software system.

Setting up a network involves conventional programming – either deriving ones own formulae for the constituent functions, or selecting them from a library. But this initial programming is hardly the same as the conventional task of deriving the complete formula that is the final implementation. Setting up the network is, in many ways, the lesser part of network derivation – it's selecting the training set as well as general network architecture that are crucial to the final implementation.

Scaling up may well represent some difficult problems, perhaps insuperable ones. If so, network programming will have to resort to modular-net implementations, perhaps hierarchically organized. In the worst case, some problems will, perhaps, not be network programmable. Large and complex software systems may be best implemented conventionally with only a module or two as a network. But none of this affects the case being made. There is no claim that network programming can, or should, replace conventional programming. There is no claim that all problems can be network programmed, nor even that all conventionally programmable problems can be network programmed – merely that some significant problems can be network programmed, and that to do so introduces a radically new paradigm for computing.

7 Network programming: a practical option?

It is of course one thing to establish that, in principle, conventional, symbolic-formula derivation is not the only way to construct machine-executable expressions; it is quite another to claim that this is not just a philosophical nicety that is being espoused, but a practically useful, alternative paradigm for computing – and it is this latter view that is being promoted. The fact that a number of useful neural-computing implementations have been developed is not sufficient to support this more ambitious case. For it could be that these are the few fortuitous outcomes from a very large number of attempts; it could be that the time and effort invested in these demonstrations is wholly disproportionate to their worth (in some practical sense); it could be that all of the neural-computing demonstrations could have been implemented more cheaply and more effectively using conventional methods, etc.

On the negative side, we might note that neither of the two champions for the novelty of the connectionist case (i.e. van Gelder and Schwarz) argue for the engineering practicality of their viewpoint. Van Gelder is quite explicit about his reservations on this possible implication of the novelty argument: "Obviously, the argument provided here does not give any *positive* reason to prefer Connectionist methods for representing complex objects over their Classical, syntactically structured counterparts. At least superficially, it might even seem somewhat perverse to design models using nonconcatenative representations. After all, one reason concatenation is so common is that having syntactic structure manifest in the representations themselves is extremely *useful* (although, as I have argued, not *necessary*) from the point of view of designing structure-sensitive operations. There are, in short, immensely appealing practical engineering considerations that argue in favor of the Classical view"(p. 382-3).

A blunt summary of Schwarz's argument about neural-net computations not being PMSs might be that neural-net computations cannot be viewed as symbolic manipulations of representations. Hence, the (in practice) non-restrictive constraint of finitude of representational 'size' in conventional computation does not carry over to be similarly 'harmless' in neural-net computations. A PMS-based computation permits the development of processing schemes (i.e. algorithms) largely unaffected by the practical limitations of actual computations. Virtually all of the finitude constraint can be packed into the representational component. This has a tremendous practical advantage. In conventional computing we can, for example, devise a processing scheme for sorting lists, and the scheme (if properly devised) is *independent of how long the list actually is.* If the sorting scheme is applied to list whose maximum length is 100, it will require very little (if any) modification to deal with lists of length 10,000.

But if we train a neural net to sort lists of 100 elements, it will, most probably, be of little use for sorting lists of length 10,000. This is because, at the moment, within neural-net computations there is no concept of processing mechanism separate from representation being processed. This lack of separation of concerns (and lack of any alternative, convenient separation) must count against the potential practicality of neural-net computation.

But note that this limitation (even if it resists advances in our understanding of neural-net computation) is not so severe as to make all potential uses of neural-net computation impractical. It may mean that to enhance a neural-net computation, a new net must be set-up and trained afresh. This is likely to be a tolerable limitation in some circumstances, especially when we note that the automatic training is orders of magnitude quicker and cheaper than manual algorithm derivation.

Clearly, establishment of the practical case can only emerge over time, but we can, however, consider a few reasons why neural computing is likely to become a useful, practical alternative. But precisely because we are considering a radical alternative the neural computing option is likely to introduce some severe distortions into the accepted software engineering paradigms. This makes foreseeing the possibilities particularly difficult, and it makes wholesale changes in accepted practice, which are naturally resisted, a distinct likelihood.

A fundamental point about neural-computing implementations is that they promise to be orders of magnitude cheaper to develop because we substitute cheap computer time training the networks for expensive programmer time manually devising and coding algorithms. A possible exploitation of this trade-off was noted above; another is described in the next section.

Within the software engineering community there is a healthy suspicion of neural computing but tempered by an acknowledgement that is does seem to work sometimes, and to some degree especially on problems that are manifest in terms of a set of input-output relationships which are resistant to accurate capture in terms of an abstract specification. The practical concerns seem to centre on validity issues which are not at all helped by the 'black box' nature of trained neural nets.

But notice that neural-net implementations are derived by a formal procedure from a well-defined set of training examples. In conventional computing the implementation emerges from a long and complicated, unkown and unknowable procedure driven by human ingenuity and intelligence. On the face of it, guarantees about the behaviour of the final implementation (and hence the specification of what is <u>actually</u> implemented) would seem to be more likely to be forthcoming from the product of a well-defined automatic process than from the outcome of the ill-defined 'manual' process.

It is no help to the network programming camp that the neural-net computations currently defy easy interpretation, but on the other hand it removes the temptation to dubiously (if not falsely) infer functional characteristics from perceived infrastructure — one of the banes of conventional computing malpractice. Perhaps the longtime dream of automatic programming, which all have awoken from since the halcyon days of grandiose AI projects, is to re-emerge in neural computing.

8 Concluding remarks

Network programming is a radically new way to compute. Acceptance of the fact that network implementations can reliably compute well-defined functions, entails a reappraisal of much of what has previously been considered fundamental to all computing. Terms like 'programming', 'algorithm', 'proof of correctness' etc. are no longer appropriate, at least, not without some considerable extension or change of their traditional meanings. These are the elements of Dijkstra's new language which is forced upon us by the acceptance of a radical novelty.

And from the terminological disruption, we can expect a methodological one. In the network-programming paradigm, constructing a reliable implementation is nothing like it used to be. Taking software reliability as an example: consider multiversion programming where a number of versions of the same problem are developed independently with the goal of achieving high overall reliability. Traditionally, this approach is only taken in cases where extreme cost (of one sort or another) is associated with software failure, simply because it is so very expensive to develop multiple independent versions of the same problem. Network programming changes all that: a network-programmed implementation exchanges programmer time for computer time – the time that the programmer spends deriving his formula is spent by a computer on the training set. Computer time is currently much cheaper than programmer time, especially when automatic training is orders of magnitude faster than manual formula derivation. Thus a multiversion network-programmed system immediately becomes more economically feasible.

Effective multiversion software also requires 'diversity' between the versions — succinctly put, 'diverse' versions have minimal failures in common. So added to the unacceptably high cost of building multiversion software, there is inability to exercise sufficient control over the programming process to be able to engineer the required diversity into the set of alternative versions. The alternative, network-programmed, versions however are easily engineered to be diverse (by systematically varying the initial conditions for training). Initial studies (Partridge and Griffith, 1994) have explored and demonstrated the reality of this phenomenon.

This opens up entirely new practical strategies for software system reliability based on populations of (sets of) alternative network-programmed versions. For example, it is both possible and productive to average the individual version outputs in order to exploit a certain type of network diversity. Dijkstra's second radical novelty, which means that approximately correct algorithms do not output approximately correct solutions, precludes version-set averaging as a multiversion-system technique in conventional software engineering (Partridge and Klyne, 1995, analyse and explore this new option).

By engineering diversity between and/or within these version sets and by using appropriate 'selection strategies' (such as majority-vote statistics or averaging) the reliability of the system as a whole can far exceed the reliability of any individual version. Notice also that multiversion software is inherently robust — no one version is crucial to overall performance. It is also an 'additive' technology: if, for example, a particular mutliversion configuration is required to exhibit greater reliability, more versions (or whole version sets) can simply be 'added' in to achieve the necessary system perfomance (see Partridge and Yates, 1995, for specific examples).

It should be made clear that not all computation that proceeds under the banner of Neural Computing is computing in terms of the new paradigm for which I have argued. Some sorts of Neural Computing are entirely conventional in nature. Typically, if the network can be handcrafted (which implies that links and/or nodes have a straightforward interpretation in terms of elements of the problem to be computed), then the resultant neural net implements a conventional computation. A key feature indicative of a non-conventional neural-net computation is that the net must be automatically trained in some way. I make no pretence to be able to delineate clearly between these two computational paradigms such that all specific models fall unambiguously into one or other of them. I offer MLPs trained with the backpropagation algorithm as representative of non-conventional neural-net computation, and I draw attention to the fact that the diverse field of connectionism, or neural computing, admits particular computational models within both paradigms.

One last, oblique reflection on this whole issue: it is prompted by Dijkstra's dismissive reaction to the artificial intelligence community. As was stated earlier: for the conventional programmer, the computer is a device that does what he could have done himself, but does it more quickly and more accurately. It's a mental prosthesis: it augments our powers of mental computation by providing crucial characteristics to supplement our brains' deficiencies. Conscious human computing appears to be symbol manipulation of the conventional type, modern computers have merely provided an extension of this same capability – an enormous extension, it is true, but nothing really new. I say 'merely' to emphasize that it is not a different sort of computational ability that's being provided in conventional computing. Computers have been designed, and the field of conventional computing has been developed 'in man's own image' of conscious computation.

This all echoes Dijkstra's dismissal of artificial intelligence as a misconceived attempt to get computers to do what people do best – why bother? But then why restrict ourselves to merely extending our mental powers of conscious computation (as distinct from unconscious computation which is, for example, what our bodies do with chemical symbols)? Why not use this powerful notion of external computation to develop mechanical computational techniques, which don't just extend our natural abilities, but provide us with a completely new way to compute?

The world of software engineering and of formal computer science needs to divert some of its attention from the development of traditional programming – i.e. formula derivation for symbol manipulation devices – to take a hard look at network programming. This paradigm can certainly be made to work, but it is sorely in need of a formal framework and a thorough analysis of how, when, and in what ways it can provide a technologically robust and reliable alternative means to compute. But this re-allocation of resources will not happen until the computer scientists really consider it to be a viable radical paradigm for computing. For, as Dijkstra says: "adjusting to radical novelties is not a very popular activity."

9 Acknowledgements

This paper is all the better for the critical comments of Antony Galton, Ajit Narayanan, and Ron Chrisley. In addition, Edsger Dijkstra kindly provided a detailed criticism of an earlier draft, and the current one is substantially better for it. It should be added that Dijkstra's assistance with the development of this article should definitely not be taken as an implication of agreement with the content. Finally, I thank the Engineering and Physical Sciences Research Council and the Department of Trade and Industry for support under their Safety-Critical Systems Programme (grant no. GR/H85427).

References

[1] Aleksander, I., Thomas, V. W. and Bowden, P. A. (1984) "WISARD – a radical step forward in image recognition," *Sensor Review*, pp. 120-124.

[2] Dijkstra, E. W. (1989) "On the cruelty of really teaching computing science," *Communications of ACM*, 32(12), pp. 1398-1404 and 1414.

[3] Fodor, J. A. and Pylyshyn, Z. W. (1988) "Connectionism and cognitive architecture: A critical analysis," *Cognition*, 28, pp. 3-71.

[4] van Gelder, T. (1990) "Compositionality: A Connectionist Variation on a Classical Theme," *Cognitive Science*, 14, pp. 355-384.

[5] Partridge, D, and Griffith, N. (1994) "Strategies for Improving Neural Net Generalization" *Neural Computing & Applications*, vol. 4.

[6] Partridge, D. and Klyne, S. (1995) "Thresholding or Averaging as Collective Decisions", Research report no. 313, Department of Computer Science, University of Exeter, Exeter EX4 4PT, UK.

[7] Partridge, D. and Yates, W. B. (1995) "Engineering Multiversion Neural-Net Systems," Research Report no. 320, Department of Computer Science, University of Exeter, Exeter EX4 4PT, UK.

[8] Schwarz, G. (1992) "Connectionism, Processing, Memory", *Connection Science*, 4, 3, pp. 207-226.

[9] Widrow, B., Rumelhart, D. E. and Lehr, M. A. (1994) "Neural Networks: Applications in Industry, Business and Science", *Communications of ACM*, 37(3), pp. 93-105.