WIPO WORLDWIDE SYMPOSIUM ON THE INTELLECTUAL PROPERTY ASPECTS OF ARTIFICIAL INTELLIGENCE

Stanford University,
Stanford (California), United States of America

March 25 to 27, 1991

World Intellectual Property Organization

Acknowledgements

This paper appeared in the Proceedings of the *World Intellectual Property Organization Worldwide Symposium on The Intellectual Property Aspects of AI*, pp. 101–119, Stanford, CA, March 1991. The proceedings are available from WIPO, Geneva, Switzerland.

An earlier version of a few short segments of this text appeared in the "Musings" column of *Chemical Design Automation News*, June 1991. The section on digital information draws in part from Pamela Samuelson's article, Digital media and the changing face of intellectual property law, *Rutgers Computer and Technology Law Journal*, vol 16, # 2, 1990. The title of the talk is inspired by Allen Newell's article, The models are broken, the models are broken, *Univ of Pitt. Law Review*, 1986, pp. 1023–1035.

1 Introduction

Intellectual property and technology have collided. Again. As has happened repeatedly over several hundred years, the creation and spread of a new technology is forcing rethinking and reassessment of intellectual property law. This time around it is software in general and artificial intelligence in particular that motivate our discussion.

If we are to make progress in the intersection of law and technology, we must understand the essentials of each. We need to understand the fundamental assumptions that underlie intellectual property law and we need to take account of the essential character of software as a technology. Most importantly we need to understand how and where the character of software may conflict with the assumptions that underlie notions of patent and copyright.

In this talk I will argue that one fundamental fact about software distinguishes it from previous technologies: programs are not only text; crucially, they also behave, and it is their behavior that is most often central to these discussions. Because they are both text and behavior, creating programs is simultaneously an act of authorship and an act of invention; programs are both engineered devices and literary works. A central goal of the talk is to make clear this important dual nature of software and to explore some of the important consequences it has for intellectual property law.

I will begin by trying to clear away what I believe to be significant confusion about the issues that face us at this symposium, by defining just a few technical concepts that are central to our undertaking here. With those concepts clearly defined, we can rephrase some of the questions that have arisen, framing them in a way that I believe will expose the crucial issues and help us make progress toward answers.

Let me note at the outset, however, than we cannot yet answer the most fundamental question: Should computer programs be copyrighted, patented, both, neither, or something in between? Grappling with that is not on the agenda today, but I believe we can still make considerable progress in setting down the proper foundation.

Instead the agenda is first, to determine what the problem is and isn't. I will claim that the essential problem does not lie in artificial intelligence or any of its variants (like neural nets). The problems we are dealing with arise instead from software in general; nothing essential would be lost if in all of these discussions we were to replace the terms "artificial intelligence" and "neural nets" with "computer program." Almost all the fundamental problems in AI programs arise because they are computer programs, not because they are AI.

Second on the agenda is exploring the character of software, attempting to make clear the assertion that programs are by their fundamental nature simultaneously both literary work and invention.

Third, I wish to explore the consequences of this view. One consequence is the unraveling of some of the abundant confusion that software has generated in the world of intellectual property. A second consequence is the realization that the now infamous litany of structure, sequence, and organization (SSO) is technically incoherent, i.e., it makes no technical sense. While the repair needed to provide coherence is not major, it is important to recognize that the phrase, as it is now understood and widely used, is technically incorrect and serves only to confuse important issues.

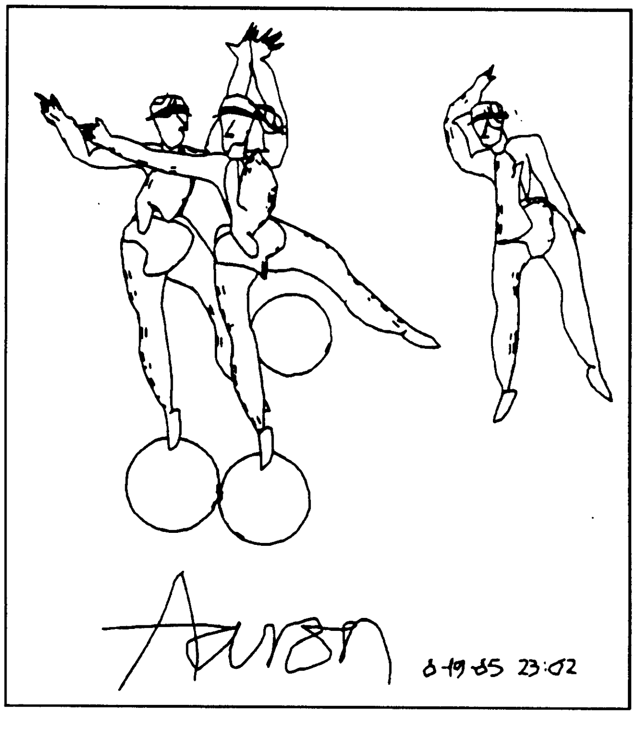
Third, I will argue that the fundamental problem is in fact considerably broader than software. It is instead rooted in the digital medium: almost all the hard issues arise in any variety of information in digital form.

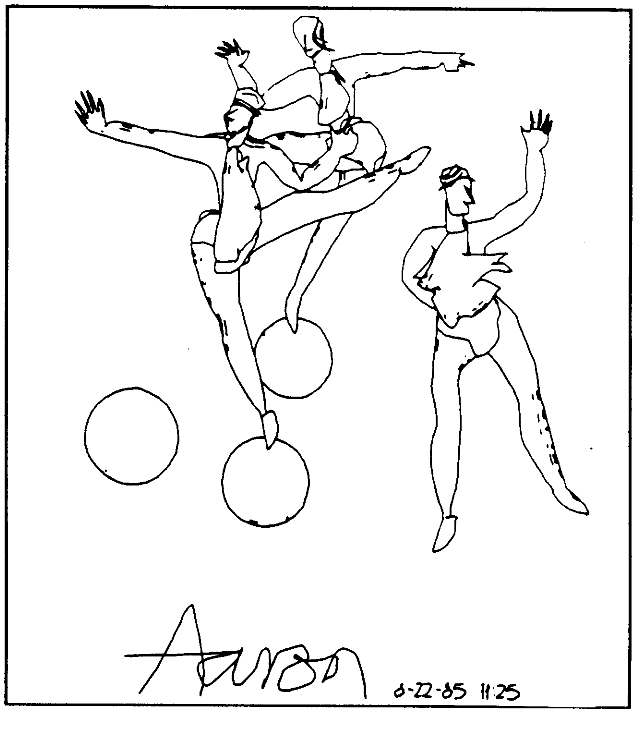
Fourth, I will propose an answer to the question raised at the outset of this symposium that gets to the heart of the issue: What do we want to protect against? I will offer a brief answer to that question, one that is surely incomplete, but one that I think will get us well started.

Finally, there remains the question, How shall we proceed? Given some understanding of software and how it strains assumptions underlying intellectual property, we have a significant problem on our hands of reducing or resolving the strains. I have some suggestions about the process, spirit, and mindset in which we attempt to do that.

2 What the Problem Is, and Isn't

Let me begin with the assertion that the problem does not lie in artificial intelligence. I will do that by showing you some work by Harold Cohen, an artist who works at the University of California in San Diego. He is an artist, but his medium is computer software: He writes programs that draw pictures. Here are two pictures drawn by a program called Aaron, early in its development.





Both of these were drawn at the 1985 AI conference in Los Angeles. Harold had brought the program with him and simply let it run during the conference, generating picture after picture like these and giving them to attendees. The first of them hangs on the wall of my office at MIT; the second I borrowed from a colleague at Stanford.

Note that despite being generated by a machine, neither picture appears "mechanical." The individual human figures appear in graceful and natural stances and the composition—the placement of figures on the page—is aesthetically pleasing. This is particularly impressive in view of Aaron's ability to generate an endless stream of such scenes, with no two precisely alike. Clearly the program is not simply reproducing stored scenes previously drawn. It is instead generating new, different variations each time, and it is doing so based on a set of principles, principles for artistic creation and aesthetics, that enable it to generate work clearly deserving of the term "creative."

This program and its results demonstrate several interesting things. First, it shows that we need not talk about what will happen *when* or *if* programs become sufficiently advanced to be creative. They already are. These pictures were generated six years ago; more recent work by Aaron is considerably more sophisticated[2]

Second, computer programs provide particularly rigorous testing grounds for our theories and understanding. A program is a set of explicit instructions that will be executed with exacting precision; hence we discover by simply trying to write a program how much (or how little) we understand a problem, and by running and analyzing the program we can get a detailed and objective determination of which of our ideas provided power in solving the problem. Hence by studying Aaron we can learn something about how creative behavior can be accomplished. They key question about the program is, "What does it know that makes its behavior possible?"

Aaron knows some things about the physical world, such as, humans have two arms, two legs and a head. It knows that human anatomy enables a certain range of motion of the limbs and it uses that knowledge in positioning the figures. It knows that maintaining balance requires putting the center of gravity over the point of support. Notice that all of the figures are well balanced in that way.

The program also knows things about the human world. It knows that there are some joint motions that are anatomically possible but are not normally used (consider the sorts of position brought to mind by the word "contorted"). It knows that humans normally adopt balanced stances, and that we do so by using our arms and legs as counter-balances. That's not the only way to balance: we might also bend from the waist to balance, but we don't normally do that. All of the figures in the picture are in normal, graceful positions, of the sort adopted by dancers, because of what the program knows about the world.

[2]The cover of the proceedings offers one example, generated by Aaron and colored by Harold Cohen.

Finally, Aaron also knows things about art and aesthetics. One fundamental principle for composition embodied in Aaron, is "Put it where you can find space." This simple, general principle turns out to be deceptively powerful. To put it to work we need to be more specific: How much space is enough? How far apart should these figures be? What kinds of overlap are allowed?

Aaron's aesthetic of composition specifies answers to these questions as well. With regard to overlap, for instance, Aaron permits partial overlap in figures, but works to avoid obscuring faces. This seems intuitively correct: having one figure obscure, say, a foot, arm, or shoulder of another seems preferable to obscuring the face. If placement of figures leaves no choice and requires obscuring a face, Aaron tries to ensure that the face is not obscured by a foot, because that suggests the unpleasant image of someone being kicked in the head.

3 The Problem Isn't Artificial Intelligence

In Aaron we have a program whose output can realistically be considered to be creative and whose performance derives from a set of explicit and clearly articulated principles for creating aesthetic works.

Hence creative programs exist. What implications follow from this?

For intellectual property I believe the implications are relatively modest. First, human action is still central to this activity: people specify the principles, then write, debug, and run the programs. Determining relative degrees of ownership in the program or its output may be problematic if there are multiple people involved in the construction and running of the program, but that is a familiar issue and one that is independent of AI and software. With human action inevitably at the core of the creative process, the ownership issues seem clear, and more significantly, they remain unimpacted by the nature of Aaron as an AI program.

As I argue in more detail below, the issues that do arise from programs like Aaron come not from the fact that they are AI programs, but from the fact that they are programs. The issues are difficult, intriguing, and are rooted in software in general, rather than in any particular variety of software.

A second significant implication of these programs is their demonstration that we understand something about creativity and aesthetics, an understanding rigorously tested by our ability to build programs that capture some part of it. In particular, we understand it sufficiently well to specify principles for accomplishing it. Those principles happen to be captured in a computer program in this case, but they came from human artists and can be expressed in English, as above. We can as a result use that same sort of understanding to instruct one another in notions of creativity. Indeed we do, as for example in books like *A Whack on the Side of the Head: How to Unlock Your Mind for Innovation*, R. von Oech, Warner Books, 1988, or *The Art of*

Creative Thinking: A Practical Guide, R. Olsen, Harcourt, 1986. Books like these and programs like Aaron demonstrate that the process is not fundamentally mysterious or impenetrable.

And what in turn is the consequence of that? It is often suggested that mankind is somehow diminished by the ability of machines to perform tasks that seem particularly human. I suggest instead that our ability to create such machines (i.e., such programs) is in fact a significant accomplishment that embellishes rather than diminishes us. The implication is not that creativity and aesthetics are any less remarkable, but that we as scientists and artists are the more remarkable for our ability to understand some parts of those fascinating and subtle phenomena. We not only understand well enough to explain them to each other (as in the books), but understand a few aspects so well that we can "explain" them to a machine, i.e., build programs that capture some of the complexities. While our understanding is surely only a bare beginning, it is still a substantial accomplishment worthy of pride, not an event that diminishes us.

4 The Problem Isn't Neural Nets

There has also been considerable discussion at this Symposium regarding neural nets and the difficulties they may pose. I suggest that for our purposes today, nothing essential is lost if we consider neural nets to be simply a variety of computer program.

A concrete example will facilitate discussion. The simple net in Figure 1 below computes the logic function known as "exclusive-or:" its value is 1 if either of its inputs is 1, and is zero if both inputs are 1 or both inputs are 0. There are five component units in the net, interconnected as shown, with weights (of +1 or -1) on the interconnections. The behavior of each unit is the same: if the sum of its inputs is greater than 0, it will emit a 1, otherwise it will emit a 0. The weights on the interconnections influence how the output of a unit propagates to other units: a 1 emitted by unit #2, for instance, will be felt as a -1 input to unit #3 and a +1 input to unit #4. If you try each of the four possible input patterns you can verify that this network computes the exclusive-or of its inputs.

More elaborate, real-world applications of networks have been used in areas like credit evaluation, where the inputs are facts about the applicant (e.g., how long have you been at your current job, do you own or rent your home, etc.), and the output is a score indicating the estimated riskiness of the loan.

For our discussion today nothing important is lost if we consider neural nets to be a variety of computer program. These programs are indeed described in a curious fashion, as a set of interconnections and weights, rather than traditional code. They are also created in a curious fashion, typically by being given a set of examples that exemplify what they are to compute, rather than an explicit description of what computation to perform. But neither the creation process nor its results are

Exclusive-Or

Inputs		Output
0	0	0
0	1	1
1	0	1
1	1	0

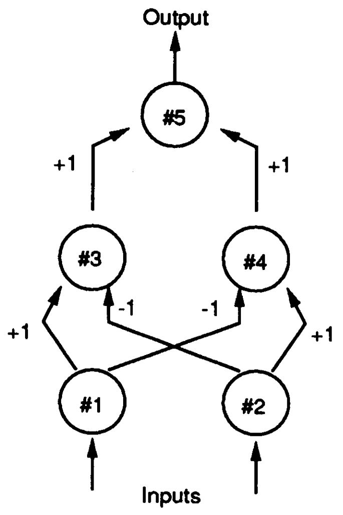


Figure 1: A simple neural network.

particularly mysterious. The creation process is a variant of interpolation, fitting a curve to a set of points; more precisely it is a form of non-linear regression. The result of the process—a specific neural net—is a specification for a mathematical function, typically considerably more complex than the $y = 5x^2 - 3x + 2$ variety we encounter in elementary mathematics, but no more mysterious.

Given a program of this sort, what issues arise for intellectual property? Other speakers have suggested that there may be questions concerning what to protect, since there is no code (at least in traditional, i.e., textual, form), and that difficulties that arise because the weights can be varied to some degree without causing significant change in the performance of the net (hence literal protection of the weights is insufficient).

I suggest that neither of these presents any significant difficulties. To protect a net we need simply protect three things: (*i*) the pattern of interconnectivity among the units, (*ii*) the weights on those connections, and (*iii*) the input and output categories, i.e., the labels that tell us what kind of numbers to put into each input (e.g., in the loan evaluation example, the length of employment), and how to interpret the number(s) that appear at the output (the estimated risk of the loan).

The first two of these can in fact easily be described in the form of a matrix, a table of numbers. The network in Figure 1, for instance, is captured in the table below. To determine the influence that unit #3, for example, has on unit #5, look in row 3, column 5. (Zeroes in the table indicate there is no connection between a pair of units.)[3]

	#1	#2	#3	#4	#5
#1	0	0	+1	-1	0
#2	0	0	-1	+1	0
#3	0	0	0	0	+1
#4	0	0	0	0	+1
#5	0	0	0	0	0

Given the ability of a table of numbers to capture the pattern of interconnections and the weights, all we need to protect is one table and a set of labels. Any real application of a net will have far more units and interconnections than the example above, but the principle is identical: the important behavior of the net is entirely characterizable by a table and a list of categories, i.e., a collection of easily described elements.

The second claimed difficulty arises from the acknowledged fact that, in any sizable network, with dozens of units and perhaps hundreds of connections, the exact weight on each link is not critical. Changes of perhaps 10% seem to have little effect on the performance of the net. In view of that, protecting the exact weights is insufficient. And if small variations on the initial set of weights doesn't degrade performance, how might we protect against someone who copies the original set, varies them randomly by a few percent, then claims independent creation?

One simple answer is already well known: we could easily employ the map-maker's trick of inserting false information into the program. As road maps often carry non-existent streets, so neural nets could be trained to display the initials of the original author when given an obscure or otherwise innocuous set of inputs. Behavior like this from a competing net would give compelling evidence against independent creation.

Hence I suggest that, at least for our purposes today, the problem is not neural nets. We may consider them to be computer programs, programs that are in fact carrying out a rather simple variety of calculation. There are interesting and difficult research problems in neural nets, including understanding how to create nets to carry out specific tasks and determining the limits on the technology. These are interesting unsolved research problems, but they are research issues in artificial intelligence. Here today we are concerned with the result of that process—the computer program that is created—and any problems of intellectual property that arise owe their origins not to the fact that it is a neural net, but to the more general fact that it is a computer program.

[3]For a good introduction supplying more detail on the general concept of neural nets and the mathematics behind them, see D. Rumelhart et al., *Parallel Distributed Processing*, MIT Press, 1986, particularly Chapter 2.

5 The Problem Is Software

The difficulties we face, and they are numerous, arise most fundamentally from the character of software as a technology. In this section I consider some of its problematic characteristics and explore how they challenge long-standing assumptions that underlie intellectual property law.

5.1 The Nature of Software: Hardware and software are interchangeable

As computer scientists learn early in their education, hardware and software are essentially interchangeable. More precisely, they are what we might call behaviorally interchangeable: any behavior we can accomplish with one we can also accomplish with the other. Any software program can be mimicked, exactly, by a piece of hardware, and the behavior of any piece of digital electronic hardware can conversely be imitated exactly by a program.

For the sake of illustration, consider a software program written to play tic-tac-toe (naughts and crosses). It is easy to write a program that will play a quite credible game, one that will win unless you choose your moves carefully. Given such a program, it would be quite simple to design a machine made from electronic components that played the identical game. That is, the behavior—the choice of moves—of each would be identical.

We could also proceed the other way around: if someone came to us with a machine that played tic-tac-toe (or any other game), we could study the physical design of the machine and create a software program that produced the identical behavior.

There would be *some* differences, of course: should we decide to change the behavior of either the hardware or software device (perhaps to improve it), we would find the software quite easy to change (typically with a text editor), while changing the hardware version would require the user of soldering irons or wrenches. It is in fact in just this sense that hardware is "hard," i.e., difficult to change, where software is "soft," i.e., malleable.

To show you that this equivalence of hardware and software is more than an abstract principle, I invite you to come to Boston to the Computer Museum where there is machine that will play tic-tac-toe with you. Two curious things about this device are that it is quite large—approximately five feet on a side—and that it is constructed entirely from Tinker Toys and string: every part in it is either a stick or a connector from a Tinker Toy set. Hardware and software are thus not only interchangeable, we need not even use electronic hardware to mimic the behavior of a program.

There are two important consequences of this interchangeability. If hardware and software are behaviorally interchangeable, the choice of which to use in any given circumstance becomes what is termed an "engineering decision." That is, the choice will have no impact on *what* we can do, only how it will get done. The decision will of course impact things like the cost of the final product, ease of modification, and its speed (hardware is often faster, but not invariably so). So we may well choose between them for reasons of cost, speed, or modifiability, but their fundamental capabilities are identical. Hence nothing fundamental changes as a consequence of switching back and forth between them, and we are free to use whichever one suits our current desires, perhaps even changing from day to day.

The second consequence arises when intellectual property law comes into the picture. While hardware and software are interchangeable in the technical world, notice the enormous difference in the variety of intellectual property protection available depending on which of those we choose. If the device is implemented in software, copyright protection will hold, yet the identical device implemented in hardware will qualify for the far stricter protection offered by the patent regime.

We have here a clash of cultures: a change of no fundamental technical significance leads to enormous variations in the level of intellectual property protection available.

This has lead historically to some interesting anomalies. For example, US Patent #4,135,240, Protection of Data File Contents, offers an improvement in the way a computer allows one user to access the files created by another user.[4] The invention is described as a physical device that works to produce the behavior desired; the diagrams and figures clearly show an electronic device, with logic gates, wires, etc.

But a brief comment in the patent is revealing: "To those skilled in the computer art it is obvious that such an implementation can be expressed either in terms of a computer program (software) or computer circuitry (hardware), the two being functional equivalents of one another.... For some purposes a software embodiment may likely be preferable in practice."

In these two sentences we have, first, a restatement of the principle noted above, and second a completely disingenuous remark concerning the embodiment that "may likely be preferable in practice." In fact no computer system to date has ever implemented this invention in hardware, for a number of good reasons, including the fact that file management is conceptually a part of the operating system, hence this invention belongs in the operating system software, where all the other file management is done. The sham of describing it as a hardware invention was necessary because the patent was filed in 1973, well before *Diamond v. Diehr* opened the door to software patents in 1981, and well before *Apple v. Franklin* established in 1982 that copyright protection applied to operating systems. Patent protection for physical devices was of course available, and by minor slight of hand an invention that is sensibly embodied in software was recast as a hardware device.

The current availability of software patents reduces somewhat the disparity in available levels of protection, but the basic point still stands: for most purposes it

[4]This is in fact the Set UID bit in the Unix operating system.

is completely immaterial whether a device is implemented in hardware or software, yet intimately associated with each of these is a very different level of protection. Software viewed as a literary work naturally conjures up a copyright approach, while hardware devices are obvious patent material. As a consequence an almost accidental and easily changed technical property of the device has an enormous impact on its intellectual property status.

5.2 The Nature of Software: Programs Also Behave

For some time computer programs have been viewed fundamentally as textual works. In US law, for instance, a program is defined as "a set of statements or instructions...," bringing to mind of course a written set of statements, and hence a textual work. Indeed a quick glance at any program reinforces this view; they certainly *look* like textual works.

This is also a seductive view, in part because we already have a body of law—copyright law—well acquainted with textual works. And it almost works: because a program is also a text, many of its text-like aspects are well served by traditional copyright concepts and vocabulary. The confluence of these two produces an almost irresistible tendency is to treat software *just* like any other copyrightable work. But this may be a case of the old saying "When all you have is a hammer, everything looks like a nail." Are programs in fact properly treated as textual works, or are we heading down that path simply because it is familiar and well-worn? There is certainly considerable traffic in this direction, as for example articles like the one by Clapes,[5] whose very title displays the mindset clearly, and whose position is a valiant attempt to see a program as fundamentally a textual work.

A program is indeed a text and many of its text-like aspects can be accommodated by traditional copyright concepts and vocabulary. But the view also causes serious problems, not because it is wrong, but because it is sorely incomplete.

A second, crucially important, thing about a program is its behavior: a program exists in order to get the computer to *do* something ("bring about a certain result"). A word processing program can for instance, insert text, delete text, copy it, move it, align the margins, etc. Hence a program can also be viewed as a collection of behaviors.

This second aspect of software is crucial: despite the longstanding view of software as a literary work, a program is not only text, it also behaves, and that behavior is an essential aspect of what it means to be a program.

This is particularly interesting because literary works—books—do not behave; behavior is normally associated with machines. Yet programs are textual works created

[5]Clapes, Lynch, and Steinberg, Silicon Epics and Binary Bards: Determining the proper scope of copyright protection for computer programs, 34 UCLA L. Rev. 1493 (1987),

specifically to bring about some variety of behavior. *Writing* programs is in fact an act of *invention.*

Put slightly differently,

Software is a machine whose medium of construction happens to be text.

Software is indeed a machine: it is as much a carefully designed artifact as any physical machine and like any physical machine it is designed to *do* something. Software is indeed a machine: the fundamental interchangeability of software and hardware shows that any program could just as well be a physical machine.

But software is also a unique variety of machine because its construction medium happens to be text.

All this phrasing is quite deliberate, chosen to emphasize assumptions underlying intellectual property law. In the US at least, we have some two hundred years of tradition suggesting a dichotomy between authorship and invention, and the clash of assumption and technology is fundamental.

5.3 Consequence: Creating software is both authorship and invention

The significance of this for intellectual property is surely clear by now: creating software is, inevitably and inextricably, both authorship and invention. It is also the first technology with this character.

To those of us in the computer science world there is nothing remarkable about this; it is in fact our everyday practice. The clash arises with the tradition in intellectual property law that authorship and invention are distinct activities, with markedly different policies attached to each. In the US the distinction has its origins in the Constitution, with its discussion of "authors and inventors" and "their *respective* Writings and Discoveries" (emphasis added). The dichotomy was given additional support by the US Supreme Court decision in *Baker v. Seldon*, which has traditionally been interpreted to mean that copyright and patent were mutually exclusive.

The distinction was largely successful: it held strong for some 200 years. But software is different, and it breaks the underlying assumption.

One consequence can be seen in the apparent contradiction that, despite 200 years of tradition that they are disjoint means of protection, software is protected by both copyright and patent. I claim the confusion is entirely appropriate. The source of the problem is not a contradiction in the practice of the law, it is instead the underlying, unjustified assumption that the two concepts are entirely disjoint.

One step further back is the underlying assumption that literary works and inventions are distinct categories. This is almost true; it is the fact that programs also behave that causes the assumption to fail.

5.4 Consequence: The same program is protected by both copyright and trade secret

Software also presents problems because it is possible to protect the same piece of code by both copyright and trade secret. When programs are distributed, only the machine code is provided; the source code is held as a trade secret. This provides the benefits of trade secret protection: no disclosure and in principle no time limit to the protection. Yet should the source code be revealed, it will be found to have copyright protection. Hence unlike a trade secret, which, once revealed, is public domain, the code is still protected.

This curious combination arises from the confluence of a number of factors, including the fact you can copyright an unpublished work, that software is routinely licensed (not sold), and that the license gives you permission to run the code, but not to read (and decompile) it. While it would make no sense to try to sell a book under the condition that it not be read, it makes fine sense to distribute code in this fashion. This defeats a fundamental underpinning that resides in at least the spirit, if not the letter of the US copyright law: the limited monopoly over reproduction is supposed to "promote the Progress of Science and the useful Arts" at least in part by publication, disseminating the ideas. Yet here we have the monopoly without the dissemination.

Once again, software breaks an important underlying assumption because it behaves. For an ordinary literary work the only way to "use" it is for a person to read it. But code can easily be used without being read by a person. As a consequence a basic mechanism—copyright—is rendered less effective in its intended effect.

5.5 Consequence: The problem of software patents

The patent metaphor similarly runs afoul of essential characteristics of software, particularly in attempts to define patentable subject matter. Historically patents have been denied to laws of nature, "mental steps," formulae, and scientific truths, partly on the grounds that such things are ideas, that ideas cannot be owned (even temporarily), and that patents are supposed to provide incentive to bridge the gap from idea to application. This works when there is in fact a considerable gap between the bare scientific truth and its useful application.

Algorithms are precisely specified methods for accomplishing a task. They are at once objects of considerable mathematical analysis and sophistication (e.g., the study of algorithms for sorting large collections of objects), and they are of immediate practical use. The simplex algorithm of operations research, the Fast Fourier Transform, and the Karmarkar algorithm for the traveling salesman problem, for instance, are all mathematical ideas, but they are mathematical ideas about how to compute something, and as such are immediately useful.

Perhaps the way out then is to deny protection, on the grounds that algorithms are indeed like scientific laws, and on the grounds that incentive for application is not needed when the application is apparent in the idea.

Yet as more and more of the world becomes describable (and described) in computational terms, more and more of what used to be invention in the physical world may get done in software, in the computational world. Computer-aided design systems, for instance, deal not only with shape but are beginning to incorporate descriptions of processes for manufacturing. It may not be long before algorithms become a widely used language for describing (and even inventing) industrial processes, e.g., a process for polishing a rough casting to produce its final surfaces. To deny patentability for such algorithms would be to deny some of the traditional incentive for creating new manufacturing processes, and only because of how it was invented and expressed, not because the process itself was any less non-obvious, or otherwise failed test of patentable subject matter.

The problem is fundamental: the patent concept relies on distinctions and assumptions that do not sit well in the world of software and digital information.

5.6 Consequence: SSO is technically incoherent

As we have noted, a program is not only a text; a second, fundamentally important property of a program is its behavior. Each of these gives us a different view of a program: we can look at it as a textual work and we can examine its behavior.

It is crucial to recognize these two views of the program are quite distinct. In particular, *they each have their own structure, sequence, and organization.*

This is reflected in a useful piece of standard computer science jargon: Programmers talk about both the *static structure* and the *dynamic structure* of programs. The static structure is the organization of the program as a textual document (the program-as-text view); the dynamic structure is its behavior, what it does when it actually runs (the program-as-behavior view).

As one illustration of how these two can differ, imagine a word processing program. Two of the things that such a program can do (two of its behaviors) are deleting words and deleting paragraphs. If we were to examine the text of that program, we would likely find in it the sort of thing sketched in Figure 2, with one segment of code that implemented the "delete word" behavior and another that implemented the "delete paragraph" behavior.

The crucial observation here is that we can quite literally switch the sequence of those two routines in the code (as in Figure 3), *but the behavior will be completely unchanged.* By that I mean that there is no way that a user can determine which of the two programs in Figure 3 he was using: If he had been using Version 1 and without his knowledge we substituted Version 2, he would never notice the difference.

This demonstrates that the two aspects of a program—text and behavior—are only loosely connected: we can make significant changes to one without affecting the

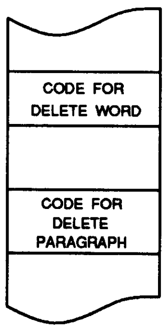


Figure 2: One structure and organization for the text of a word processor program.

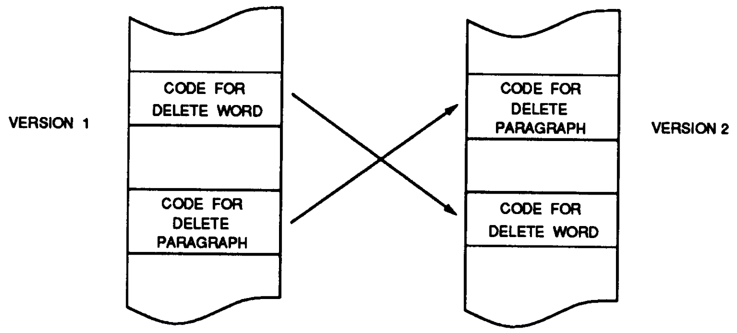


Figure 3: Changing the structure and organization of the text can leave the behavior unchanged.

other. Hence the structure, sequence, and organization of the *text* of a program is distinct from the structure, sequence, and organization of the *behavior* of the program.

Another way to illustrate the independence of these two is to imagine using the word processing program long enough that you become quite familiar with the behaviors it offers. Imagine being in the midst of using the program and saying to yourself, "I know very well what this program can do, but I wonder what the code actually looks like?" What might you be able to tell about the *text* of the program from your experience in using its *behavior*?

Essentially nothing.

Where, for instance, in the program text is the code for the delete-word command? You can't tell. Does it appear before or after the code for the delete-paragraph command? You can't tell. Hence you may know the structure, sequence, and organization of the behavior of the program quite well, yet be able to draw almost no conclusions at all about the structure, sequence, or organization of the text of the program.

Program text and program behavior are thus only loosely connected, each with its own structure, sequence, and organization.

Note that this is not an artifact of the example we have chosen. It is in fact quite common and arises in part as a consequence of the standard programming practice of dividing programs into independent subroutines.

The independence of program as text (static structure) from program as behavior (dynamic structure) has several significant consequences. The first consequence is that it makes no technical sense to talk simply about the "structure" of a program, because the term is ambiguous and the distinction matters (the dynamic structure can be different from the static structure).

Second, with all due respect for an otherwise carefully argued position, *Whelan v. Jaslow*[6] runs afoul of this distinction: The (by now infamous) litany of "structure (or sequence and organization)" is flawed by the failure to distinguish between the static and dynamic views of a program. As a consequence the terms are inherently ambiguous, in the important manner noted above.

Indeed, the decision trips coming out of the starting gate, in note 1:

> 1. We use the terms "structure," "sequence," and "organization" interchangeably when referring to computer programs, and we intend them to be synonymous in this opinion.

This is, alas, exactly wrong technically: it is precisely because a program is not only text, but also behaves, that these terms (in all their meanings) are not synonymous. The problem is thus compounded: where the terms were merely ambiguous, they have now been declared equivalent when they are not. The sequence of behaviors can be quite different from sequence of instructions encountered in the program text (recall our ability to change the order in which the subroutines appeared in the program text). The structure of the behavior is quite distinct from the structure of the text (the behavior stayed the same even when the text was reordered).

References to, and hence questions about, "structure, sequence, and organization" as used in *Whelan* are thus inherently technically defective and as a consequence often unanswerable. Legal use and future decisions may yet provide workable definitions for these terms, or supply better alternatives, but it is important to understand that these currently make no technical sense.

[6] *Whelan Associates v. Jaslow Dental Laboratory,* 797 F.2d 1222 (3rd Cir. 1986).

6 What Do We Want To Protect Against?

The question has been raised several times in these discussions, "What is it we want to protect against?" As one attempt to answer it, I suggest that

> A qualifying work of software should be protected against the trivial acquisition of functional equivalence.

It is, first of all, functional equivalence that matters, because the fundamental value in a program lies in its behavior. The question is not whether the code in two programs is identical, or whether the weights in two neural network are precisely the same; the key is instead behavior. Do the two competing programs *do* precisely the same thing? If so, then one is a complete substitute for the other and hence it doesn't matter which I use. Thus if you can, in some fashion, take the behavior you have taken the value.

Second, it is the trivial acquisition of functional equivalence that matters. This is important because trivial acquisition means the innovator will have no lead time in which to benefit from the creation, and loss of lead time will then likely lead to loss of incentive.

Third, note that "acquisition" is purposely vague: *how* you accomplish it is not particularly significant. The focus here is not the means, it is instead the goal, the valuable goal of equivalent behavior. Acquisition may be by straightforward piracy (i.e., direct copying of either source or object code), it may be via reverse engineering, exhaustive inquiry (in the case of neural nets or data bases), or by any other means now known or yet to be invented. It matters rather little how a functional equivalent is created if the effort involved is trivial.

Hence I suggest that we have the beginning of an answer to the question of what to protect if we focus on functional equivalence as the key property, and recognize that trivial acquisition is the issue, independent of the method by which it is accomplished.

Framing the answer in this fashion has several benefits. It focuses the discussion on the important property of programs (behavior) and the important result of copying (functional equivalence), rather than on the medium of creation (e.g., code, neural nets, or databases). It focuses discussion on the situation we want to avoid (trivial acquisition of equivalence), rather than on the means that can be used to bring about that situation. In focusing on the situation rather than the means, it obtains a degree of technology independence. Reverse engineering, for example, is currently largely manual and hence sufficiently difficult that it does not enable trivial acquisition of any sizable program. Under the suggested principle then, reverse engineering (e.g., "clean room" style re-implementation of a program) would be considered acceptable. But as automated tools for reverse engineering continue to improve, we may someday reach a point where the effort does deserve to be labeled trivial, and at that point that approach may appropriately be judged off-limits. By focusing on the situation, rather than the technology, we have a chance of creating principles that are not soon outdistanced by technological change.

Finally, framing the issue this way also focuses the discussion back on the fundamental issue of incentive. No matter how elegantly written, our principles and laws will be of little use if they lose sight of the goal that brought us to these discussions.

7 The Problem Isn't (Just) Software

Earlier I suggested that the fundamental problems we face in this symposium do not arise out of artificial intelligence, but out of the status of AI systems as programs. In fact the problem is even broader than that.

The problems we face arise most centrally from the digital medium, i.e., any information in digital form. The problems arise here because information in digital form has a number of remarkable properties, properties that are at odds with some of the fundamental assumptions that support enforcement of intellectual property law.

Any information in digital form, is, for instance, orders of magnitude easier and cheaper to replicate. Floppy disks are treated as pieces of paper, yet each of them holds the equivalent of 1500 printed pages and requires only seconds to copy, at close to zero cost.

Digital information is trivial to distribute, worldwide, almost immediately, at very little cost, over an existing infrastructure (i.e., phone lines); we need no additional distribution channels.

Digital copies are *indistinguishable* from the original; no other medium has that remarkable (and troublesome) property.

Digital information has a breadth of descriptive power we are only beginning to explore: we use it to capture not only software and text, but music, photographs, speech, shape descriptions (e.g., computer-aided design systems), and who knows what else.

Digital information is easily used by multiple people simultaneously: one copy of a file on a large computer can be read by hundreds of users at once and accessed by tens of thousands of people over time at no additional expense.

Digital information is orders of magnitude more compact: a compact disk holds 500 Megabytes, the equivalent of 500,000 printed pages. The information formerly held in 100 cartons of paper has been reduced to the size of something you can easily slip in your pocket.

All of this undermines assumptions built into the enforcement, if not the principle of intellectual property law. What do we do when a wide variety of expensive and complex finished products be duplicated *exactly*, at effectively zero cost, and distributed almost instantaneously around the planet? When there is no degradation in successive copies, the replication can continue unabated. When copying and distribution is nearly free, there is little to keep the value from slipping out of our hands. When hundreds can read the same copy, there's no need to buy more than one.

Simply put, *for other forms of intellectual property, physical law tends to support intellectual property law.* The mere weight and volume of the objects copied, the need for distribution channels and physical transportation, etc., all stand in the way of the intellectual property thief. All of that is missing in the case of information in digital form.

The problem isn't (just) software; it's digital information.

8 Designing the Future

What can we do in the face of all this? I suggest that it is crucial to proceed in a non-traditional fashion. Rather than asking "What does the law say?" we need to begin to ask "What *should* it say?" Put another way, we need to begin treating this as a design problem.

In its simplest terms, a design problem is characterized by a goal, a blank sheet of paper, and some tools. First and foremost is the goal: we must know what we are trying to accomplish. Second is the blank sheet of paper: initially at least, there are no constraints on the solutions we can entertain. Third, the process is driven by the goal, not by the tools. The primary question is not "What do we know how to do?" but "What do we want to accomplish?"

I believe we also need to think of this as a problem of industry design. Any decision about intellectual property law is also a decision about the grounds for competition, and will thereby inevitably shape the future of the software industry. Narrow protection (e.g., protecting only literal source code), for instance, would permit more latitude in creating functionally equivalent programs. This in turn would lead to a marketplace with more software clones and likely shift the basis of competition from functionality to other factors like price, quality of implementation, user support, etc. Broader protection, by contrast, would likely lead to an increased emphasis on innovative functionality in second comers: being competitive would require functionality sufficiently better that it could win converts from established alternatives.

All this depends on the maturity of the industry as well: early on in most industries there are many small competitors and many new customers. As the industry matures the tendency is toward fewer, larger, well-established companies seeking to lure customers from each other. The best rules for the game thus may differ depending on the stage of industry development.

Hence we are not only attempting to design the future, we are aiming at a moving target. The problem is quite difficult, but I believe there is considerable utility in conceiving of it as a design problem and conceiving of it in terms of industry design.

Consider, for example, the problem of patenting algorithms. Arguments in the legal community have dealt with the somewhat metaphysical problems of the meaning of the term "process," the nature of mental steps, whether algorithms are scientific truths and so forth. We may in fact be better off asking simply, What would happen to the software industry? Would we be better off with or without patents? If we allow software patents, would, as some people claim, software innovation be crippled because of the difficulty of finding all the relevant patents when creating a new system? If we disallow patents do we reduce to any significant degree the incentive for creation of or investment in new ideas? Perhaps the question is in fact better asked in these terms, as one of industrial policy, economics, or psychology, rather than metaphysics.

A similar approach can be take to the question of the scope of copyright protection. Rather invoking the near-mystical principle of idea vs. expression (which even Judge Learned Hand admitted was "inevitably *ad hoc*"), we might phrase the question pragmatically in terms of lead time: If the time between new releases of a product is typically shorter than the time required to reverse engineer it, the industry will to some extent take care of itself, if we require nothing more than independent creation.

9 Summary

I have argued that artificial intelligence is not currently a significant source of problems for intellectual property law. I have claimed instead that the interesting problems arise from software in general and can only be solved by a clear understanding of the character of software as a technology. One crucial part of that understanding is that programs are not only text, they also behave; software is a machine whose medium of construction happens to be text. Creating programs is as a result simultaneously a work of authorship and a work of invention. In crossing that allegedly unbridgeable barrier software creates significant conceptual difficulties, conflicting as it does with assumptions that have long been a part of the legal system.

The problem is also larger than software alone: digital information of any variety presents intriguing difficult problems for our current intellectual property law framework.

We stand at a uniquely exciting time: information is going to be the key commodity of the next century. We are faced with both the opportunity and the necessity of designing the groundrules for that future.

It is in this sense that we need to put aside, even just temporarily, the vast and often valuable experience we have with several hundred years of intellectual property law. For a few moments, at least, we need to examine this problem free of two hundred years of established concepts, categories, and assumptions. In the end it may turn out that some or all of the existing mechanisms of copyright and patent will (once again) prove sufficiently adaptable to gracefully accommodate even this technology. But we need to know that those are the correct tools, not merely that they are the available tools.

And perhaps they will not prove sufficient. Software and digital information generally are unique forms of technology. That uniqueness may have an unavoidable impact on intellectual property law: its clashes with assumptions underlying our current practice may be so profound that some new form of protection will be required. We must have the opportunity to face this question free of preconceptions and the chance to design the future rather than forcing it into molds that were designed in the past.