

When Is a Work-Around? Conflict & Negotiation in Computer Systems Development

Neil Pollock
University of Edinburgh

The notion of a “work-around” is a much-used resource within the sociology of technology, reflecting an interest in showing how users are not simply shaped by technologies but how, through adopting artifacts in ways other than those for which they were designed or intended, are also shapers of technology. Using the language and concerns of actor-network theory and focusing on recent developments within computer-systems implementation, this article seeks to explore and add to our understanding of work-arounds through unpacking the work of one group of “users” as they attempt to tailor and rollout a system within the administration departments of their university. This article argues that paying attention to the various networks that lead to and from work-arounds can improve our understanding of the way users both shape and are shaped by technologies. Focusing on work-arounds as “networks in place” also allows us to highlight some of their contingencies; for example, the other actors and entities on which these depend and are constituted.

Keywords: actor-network theory; programmers; software; users; work-arounds

Introduction

What is a “work-around”? Typically, the concept is used to explain how one actor is able to adjust a technology to meet their particular needs or goals. Indeed, one of the most significant analyses of the practice of work-arounds appeared in the work of Les Gasser some years ago; Gasser describes work-

AUTHOR’S NOTE: I would like to thank all those at my research site, particularly the programmers in Administrative Computing Services (ACS). I would also like to acknowledge the support of the UK Economic & Social Research Council (ESRC), under which this research was funded. The actual research was initially carried out with an ESRC studentship and later developed while

Science, Technology, & Human Values, Vol. 30 No. 4, October 2005 1-19
DOI: 10.1177/0162243905276501
© 2005 Sage Publications

arounds in relation to the ad-hoc methods deployed by users of administrative computer systems who were attempting to fix problems or glitches in their work. Gasser wrote that working-around means “. . . intentionally using computing in ways for which it was not designed” or avoiding a computer’s use and “. . . relying on an alternative means of accomplishing work” (1986, 216).¹ Sociologists in general, and sociologists of technology in particular, continue to be fascinated by the practice and process of work-arounds. This, it might be suggested, reflects a wider interest in showing how users are not simply shaped by technologies but how they are *also* shapers of technology. Put another way, the term is often a useful trope to emphasize the differences between the “logics of a technology” and the “logics of human work” (Berg 1998), with the actual practice of work-arounds highlighting the effort necessary to bring these two factors into line. Gasser, in this sense, can be read as an account of how actors, through deploying some form of effort or skill, are able to overcome a difficulty or a constraint imposed by a technology. A stronger version of this argument is perhaps Bryan Pfaffenberger’s (1992) description of the “technological adjustments” carried out by users when a new production process or artifact is introduced into their work setting. As Pfaffenberger sees it, the users, rather than accept the discipline of the new system “. . . engage in strategies that try to compensate for the loss of self-esteem, social prestige, and social power that the technology has caused” (1992, 286). Typically, then, the common understanding of work-arounds is clear and unambiguous; they represent resistance on behalf of users and the means by which they attempt to wrest control *back* from a technology or an institution.

What motivates this article is the way in which work-arounds have become a much-used resource within the sociology of technology but, with a few notable exceptions,² as a topic they remain for the most part surprisingly underinvestigated and theorized. What is often missing in many discussions is any reference to their genesis or outcomes other than these general notions that users *also* shape technology or that work-arounds correct a misalignment between a technology and the desired goals of its users.³ In contrast, I argue that a reappraisal of the term is both important and timely for two main reasons. First, as computer systems, the technology discussed in this article,

I was working under the ESRC’s Virtual Society? Programme. Finally, I would like to thank Mike Michael, Chris Stokes, Chris Ivory, David Edge, Luciana D’Adderio, and James Cornford for their comments on earlier versions of this article. Please address correspondence to: Neil Pollock, School of Management, William Robertson Building, 50 George Square, Edinburgh, EH8 9JY, Phone (work): +44(0)131 6511489, Phone (home): +44(0)131 2281034, Fax: +44(0)131 6683053, E-mail: Neil.Pollock@ed.ac.uk

spread ever more widely and into increasingly diverse and new domains, powerful incentives for increased standardization are brought with them (cf. Agre 2000). This leads to inevitable tensions about which elements in these settings should be standardized and which elements should not (cf. Star & Ruhleder 1996). An analysis of user work-arounds remains an essential part of understanding how such “misalignments” are reconciled.

Secondly, in fields such as management and administration computer systems development, there is a blurring of the once clear distinction between users and producers of technologies. Increasingly, many systems are designed and built so that they are customizable by their users (Brady et al. 1992), meaning that users also engage in the construction of these technologies. The upshot is that it is now increasingly difficult to say exactly who has responsibility for the final shaping of systems and their implementation (cf. Suchman 1994). In this context of changing and less determinate technical divisions of labor and responsibilities, there is a need for analysis that puts the user and their modifications, as well as the ambiguity surrounding the process, at the center of its concerns.

The aim of this article, then, is to reawaken our interest in the topic of work-arounds in light of these new and more complicated technological practices. To try to do this I present the example of a group tasked with the job of customizing and implementing a “prebuilt” management information computer system (known as MAC) within the centralized administration departments of a university. Modifying technology is a routine and necessary aspect of this group’s work, although they often find that some of their work-arounds promote tensions between them and the original designers of the system. Below I attempt to develop a basic understanding of some of the factors that lead to these work-arounds—what I am calling “networks in place”—as well as some of the tensions that lead from them. A backdrop to this study is work stemming from the actor-network approach, particularly Madelaine Akrich’s (1992) important article on how technologies embody “scripts.” In this first part, I review this article as well as make some suggestions about how we might adapt and deploy this form of thinking.

The Designer-Script-User Approach

In one of the most cited articles in the sociology of technology, Madeleine Akrich’s (1992) describes how designers, when building technologies, also build “scripts” into those technologies. Users, she argues, once they take-up and use a technology, can then be seen to be enacting a script,⁴ though she is careful to point out that scripts are never enacted straightforwardly, as users

will often perform work-arounds, or what she calls “mechanisms of adjustment,” to modify an artifact (and script) to more closely fit their particular circumstances. To work through this concept, she discusses the design and use of a photoelectric kit that was providing electricity to a village in French Polynesia. She outlines how the photoelectric kit suffered from one major problem: when electricity was most in demand the kit was apt to break down. The power outage occurred because it was possible to damage the kit if it was allowed to run down, and engineers, assuming the users would be unable or unwilling to properly maintain the kit (i.e., the script), installed a “control device” that would make the kit inoperable. However, as the control device was continually braking the circuit, residents would call upon the local electrician, who, tired of receiving calls late into the evenings, eventually installed a “fused circuit” in parallel with the control device. This meant that when the power was cut off, the users could bypass the problem themselves by using the *new* fuse.

The key issue for consideration here is that if we are to accept that users play out scripts when using technologies, how are we to understand modes of use that deviate from the script? Are they simply a result of “other” scripts, the agency or skill of people, or something else? One reading of the Akrich paper is to say that the problem is posed at the level of a *choice* or a dilemma. The electrician can either let the users live with the technology as designed and succumb to its prescription (i.e., have the inconvenience of constant power interruptions) or s/he can install a fuse, but this might be to risk straining relations with the designers of the technology (the electric company). In other words, the suggestion is that the electrician, through deploying his/her skill, is able to exercise some form of “discretion.”

As already suggested, the danger is that without unpacking a work-around and looking at *what leads to and from work-arounds*, it is possible to read the situation as the electrician having control over the situation and being able to decide on possible outcomes and bring these about. Moreover, it can also be read that all of this will happen at the expense of certain other entities and actors (i.e., the electrician is wresting some form of control back from the technology or the electric company).⁵ In contrast, Mike Michael (1996) makes the appealing argument that just as we can describe a technology as prescribing one form of use, *perhaps* the same technology might also incorporate a script that enables its abuse. A technology does not embody simply one script, or order, but, according to Michael, they can embody “multiple” scripts.⁶ Moreover, these multiple scripts can often be contradictory, meaning, that just as a car, for instance, can demand a certain form of use (i.e., safe and careful driving), it can also enable the reverse (i.e., in the case of “road rage” it can be used to intimidate other drivers). While an interesting

argument, it raises some further questions: if a technology does embody multiple or contradictory scripts, then why are certain uses more likely than others? Why, in the main, do car drivers follow the “safe and careful” form of use? Is the user disciplined toward one role over the other? Seemingly, yes, or at least this is what Bruno Latour (1992) argues that engineers “bet on” when they attempt to anticipate the desires or goals of their users. Latour writes that this way of counting on earlier “distribution of skills” to help narrow the gap between “built-in users” or “users in the flesh” is like a “*preinscription*” (257). In short, what he is suggesting here is that the tendency toward one form of use is already present in the wider network. Another method of describing such networks might be to talk about a “network in place.”

In the following discussion of work-arounds there are aspects from the above that I want to take-up and develop: these are Michael’s concern for “multiplicity” and what I am calling, after Latour, a network in place. The argument is that the tendency for work-arounds is already present in the networks that those implementing the computer system (MAC) inhabit—these networks in place. Having established this, I then analyze these networks by considering some of their “contingencies”—for example, the other actors and entities on which these networks in place depend and by which they are constituted. In particular, I examine their connections to the “original” designers of MAC and the computer system itself. Both are pivotal actors who simultaneously demand and promise the possibility of work-arounds and major obstacles, questioning and hindering the progress of the implementation.

One final clarification is needed before we turn to the empirical material. It will have become apparent that I have been discussing the language and concepts associated with the “use of a technology” and that I am attempting to apply this to an example that is normally thought to be one of implementation. I think there are good reasons for doing so. Namely, as has already been suggested, designers and users are not well bounded. Mackay et al. (2000), for instance, argue that the conventional distinctions between production and use cannot always be applied to information technologies, as users are becoming more like producers.⁷ We might, perhaps, advance this argument in the other direction and suggest that, just as the notion of the user was found to be more complex than was traditionally assumed the case, it might be suggested that producers also increasingly play contrasting roles. For instance, a technology like MAC is not reliant on one set of clearly defined producers delivering a system to a user but on an extended network of computer professionals working in and for different organizations. The group implementing MAC at the local site, for instance, was made up of people with various level of skill and expertise, ranging from those who had experience with similar

implementations elsewhere to those who had been recently seconded in from nontechnical roles in other parts of the university. This group found that they were one element in this long chain and that they were tasked to work with the system in a certain way; this was linked to the efforts of the original designers of the computer system to ensure that MAC's code was modified only in the ways *they* deemed appropriate. In other words, the designers were attempting to configure the local programmers as their "users."⁸ Indeed, as Friedman describes, hardware and software suppliers often think of the computer-system developers to which they sell products as their "users." It is clear, then, that the meaning of "user" is shifting as the nature of computing itself changes (Friedman 1989; Mackay et al. 2000). Indeed, for Suchman, the key is to deconstruct such simple terms as "designer" and "user" and, at the same time, bring to the fore the relevant social relations that cross the boundaries between these two groups. In this sense, it might be suggested that work-arounds represent one aspect of the relations between these groups, as well the means by which the producer/user boundary is constituted. This leads us into an examination of the MAC system.

MAC and the Delphic Oracle

The oracle of Apollo at Delphi that gave answers held by the ancient Greeks to be of great authority but also noted for their ambiguity.⁹

The material produced here is from an ethnographic study carried out at one of the university sites where MAC was being implemented. Indeed, the MAC exercise involved most universities in the UK, as the system resulted from a decision by the centralized Universities Funding Council (UFC) in 1988 to "... take action to meet the increasing need for more and better management information systems in universities" (Goddard and Gayward 1994, 45). The idea was that the "... cost would be reduced substantially by universities working together to develop new systems common to all" (ibid., 45):

The UFC therefore established the Management and Administrative Computing (MAC) Initiative, a unique attempt to transform administrative computing across the whole university system. The initiative was placed under the control of a MAC Initiative Managing Team and all institutions were to be brought into cooperative groups (called Families) with the aim of all members of each Family eventually using the same administrative computing software and jointly developing and maintaining it (ibid., 45).

The original designers and builders of the computer system (hereafter, the Designers), which was implemented at the site where I did my study, work for one of the world's largest software organizations, Oracle (hereafter, the Technology Vendor). In order to manage the implementation, the universities created a company called "Delphic" that was directly responsible for liaising between the Technology Vendor and each of the sites. While several of the people that I worked with, especially those who had spent time at the Technology Vendor on behalf of Delphic, pointed out that "frictions" existed between the Programmers employed by the universities and those working for the Technology Vendor, it might also be suggested that the word "delphic" is an accurate description of this relationship. Most of this ambiguity existed around the so-called "80/20" rule. By this it was meant that the system was something of a "grey box" (cf. Fujiumura 1992): the design and building of the bulk of the system, the responsibility of the Technology Vendor, would be then delivered to each of the sites. Importantly, however, a small part of the systems was left to the discretion of computer Programmers working at each of the universities, who—working in close relation with the Designers—would attempt to "tailor the system" to the specifics of each of the sites. The boundary between the 80 and 20 and this tailoring work was the focus of my study.

The Delphic Support Desk

I spent several months working with one group of Programmers hoping to understand just how they managed to get their MAC system to work. One of the most intriguing things about studying this group of Programmers, and something that I had barely anticipated before I started the research, was that they would sit for hour after hour in front of their terminals, barely uttering a word. To ask them a question would be seemingly to break their concentration with the machine, to disturb the peace of the office. Even when sitting inches away from them, I was to learn nothing about the implementation. Nonetheless, the longer I was there, it seemed, the more they got used to me. And after a while, I found that they would every now and then stop working to tell me something about what they were doing, something about the code.¹⁰

Much later, however, I would realize that even while we had sat there in silence, they were in fact speaking, sometimes *shouting*. Their method of communication was electronic mail. It was this realization that they were in fact talking in the main via e-mail (Sometimes even preferring to e-mail the person sitting across from them!) that led me to begin to sift through old, archived messages. One particularly interesting source was something called the "Software Problem Bulletin," which was a sort of online help desk or

Problem Log run by the *Delphic Support Desk!*¹¹ The Programmers used the Problem Log as a type of “last resort”: if they were unable to resolve difficulties concerning MAC within their own local communities, then they reported the problem to the Support Desk, who either suggested a possible solution or passed the message as a possible bug to the Technology Vendor. The Technology Vendor would respond to each message by appending their comments (i.e., their answer to the problem). The Log was available to all the Programmers at the various sites, and they too would often post suggestions in reply to a message.

Comprising some several thousand e-mails, the Log reads like a working history of all the steps taken so far on the project. I had heard the term “work-around” continuously from the moment I first became involved with the Programmers, and the word appears in the Log. I ran the FIND facility on my word processor as a method of giving me access to other discussions about work-arounds. The first message found was a description of a problem. Over the Easter period, Carole, one of the Programmers working at a site, had attempted to install the latest release of the MAC system, version 1.4, just released by the Technology Vendor. At the same time, she attempted to upgrade the software platform that MAC would run on, Oracle 7.1.3. However, there was a problem: MAC 1.4 cannot be loaded onto Oracle 7.1.3. According to the e-mail, a small program called BuildMAC, written by the Technology Vendor to assist in such upgrades, will not perform as it should. The message goes on to mention how a similar problem was reported at one of the other sites some months earlier. A Programmer called Liz had been attempting the same process and, as for Carole, the BuildMAC program had not carried out the upgrade.

Intrigued by the discussion being carried out on the Problem Log, I continued to search the postings, hoping to understand more about the genesis of this problem. Seemingly, it had begun when Liz had written to the Delphic Support Desk, describing her difficulties, and was told by one of the Programmers that

Liz, unfortunately Oracle 7.1.3 is unsupported against all MAC software currently released, so these problems cannot be reported as bugs to [the Technology Vendor] but they may like to have the problems passed on for “information purposes only” to help them prepare MAC for Oracle 7.1.3.

In responding to this message, Liz pointed out that when they first ordered Oracle 7.1.3 they did “ask” the Technology Vendor which version would be most suitable, and they were told that their choice would be fine. The matter was not mentioned again in the Problem Log, and despite the fact that MAC is

not supported against her particular software platform, *Liz attempted to modify BuildMAC by reworking its code*. Moreover, once the work-around was complete and Liz had loaded the new version of MAC, she posted the rewrite to the Log as information for others. I will develop this discussion in a moment, but first I want to consider a different issue: what can be said about the mode of use of the Programmers—their attempts to modify the code?

The programmers at the site where I carried out my study defined work-arounds as a necessary and important aspect of their routine work. “Things” would never quite fit or be the way they should be. Often, a feature of the system would be too complicated for the end users, or one aspect of the new system would not work with the existing software infrastructure. Such problems require innovative fixes or the rewriting of code. Consider the following diary extract of a conversation I had with someone called David:

There are a few problems with loading the data into the system and David says: “It’s OK, I’ll work-around it.” David continuously talks about work-arounds. I laugh and say to him, “Another work-around. It seems to be all work-arounds here.” “That’s life,” he replies (a little dryly).

To program is to perform work-arounds, to bypass constraints, and to rewrite code. In other words, we might think of these Programmers—and, indeed, it would be in keeping with how they think of themselves—as *bricoleurs par excellence*.¹²

The image I want to develop here is of people drawing on past, or existing, knowledge, experience, or skill to confront their current situation and problems. Thus we might understand these constant attempts to work-around the code as the “networks in place” of these Programmers. This is partially in keeping with what was suggested earlier: that the tendency toward one form of use is already present in the wider networks of the user, and this is what engineers “bet on” when they attempt to anticipate the desires or goals of their users (Latour 1992). Of course, the crucial aspect in understanding these networks in place is to focus on their contingencies (i.e., the other networks on which they depend and are constituted).

When is a Work-around?¹³

Returning to the discussion of Liz and Carole, what is important to note, in terms of the argument being developed in this article, is that while Liz is attempting to rewrite BuildMAC, she receives help from her colleagues across the other sites *and* the Designers at the Technology Vendor. In an earlier message, for example, Liz describes some of this collaboration:

Thus investigated with [the Technology Vendor] how to get BuildMAC to use ProC1.6 and pick up the include files from sqllib/public. [They] initially suggested renaming executables and using links, but wanted a proper way, so—amended [their] standard .mk files (sqlmenu5.mk srw.mk sqlforms30.mk) changing the default ProC make file variables from 2.0 to 1.6 as follows . . .

What is interesting about this is that Liz’s work-around is seemingly “legitimate.” In fact, it is a necessity if her system is to ever work. Here, the work-around—changing the default ProC file variables from 2.0 to 1.6—*is use*, and we view the Programmers and Designers as colleagues discussing possible solutions. Work-arounds are very much part of the work of implementing a system, or, “that’s life,” as David from the office puts it.

Several weeks later, however, one of the Designers at the technology vendor appended the following statement to Liz’s message, essentially rejecting the recoding work that she did:

. . . thank you for supplying this information. Unfortunately I am forced to close the bug as rejected as this is the only state applicable as this code was not released for that version of the PRO*C compiler.

Despite the fact that the technology vendor did not support Liz’s rewrite, Carole went on to use this solution when she encountered the same problem some time later. Yet, Carole’s work-around was not so straightforward: she was unable to get the “PRO*C compiler” to work, and she is forced to ask the technology vendor for help. Some days later, one of the Designers posts a message to the Log, describing Carole’s problem:

I mailed Carole to ensure that it was the v1.6 PRO*C compiler that was being used. It was. On further investigation by our DBA [Database Administrator], and after some consultation with Carole, it would appear that a patch applied to the 1.6 PRO*C application is the cause of the problem.

Here, the Designer identifies the problem as being with Carole’s use of Liz’s work-around (a “patch” applied to the 1.6 PRO*C application). In a further message to the Log a few days later he summarizes the situation in the form of a final report to the Delphic Support Desk:

As you may know, [Oldcastle University] migrated from [MAC] 1.3 to 1.4 last week and encountered some problems which we helped with. We also advised them to migrate to 1.5, as 1.4 was no longer supported. This they did over the weekend and again had some problems, which I have mentioned in the log. They contacted me on Monday morning and I have been looking at the problem(s) over the last day and a half. We have carried out a few checks and offered

some advice on overcoming some of the problems, but it would appear that the problem lies in the data that they are working with and not a problem in any of our code . . . Quite simply, I cannot justify any more time on this problem as it does not appear to be a problem with our software, rather a problem on site which may well require a great deal of time to identify . . . Their current work-around is to use the basket 4 forms against the basket 5 database. I have expressed my concern over this and warned them that this is unsupported, but they appear to be confident that they have an adequate work-around.

Sometimes work-arounds are not considered normal working practices. If we were to think of an image of a network in place we would see how the Designers, with sleight of hand, begin to disrupt this network. The Designer is not performing the “collegiality” that we saw before, but is attempting to establish *difference* (i.e., to reconfigure the Programmers’ relationship with MAC). To glance at the network now, we can catch sight of other networks coming into play, flexing and pulling to create real distance between the modes of use: *now* it is easy to see “when” the mode of use is a work-around and when it is something else.

To summarize this section, these practices are proscribed because as the Programmers carry out their modifications they call into question the Designers’ responsibility toward MAC and thus the distinction between just who should be doing what. In other words, either they infringe on an important part of the code or they combine, or bricolage, in ways the Designers do not like. At the same time, however, work-arounds are demanded by the Designers in order to tailor the technology to the specifics of each of the sites (to work with existing software platforms). Importantly, it would seem the Designers of the system “bet on” the skills of the Programmers to carry out such modifications. So, one aspect of the contingency of these networks in place is that they are reliant on, and constituted by, this ambivalent situation where work-arounds are both problematized *and* supported by the Designers—what might be called the *tension of work-arounds*.¹⁴

Reconfiguring MAC: the Skills of the Programmers

A further aspect of these networks is that they are reliant on the efforts of the Programmers and their skill in working with the code. Such a relationship is not, as you will see, a straightforward one. Sima, for example, one of the other Programmers who worked in the office with David, sat frustrated for weeks attempting a (small?) work-around on a “printer script.” Sitting opposite her, I listened to her frustration as she talked to her computer, urging the program to compile. She was telling me how in her sleep at night she would

even dream of the problem, constantly working through the code in her head, taking her thoughts down the different paths, following what was, to her, the essence of the code as it made its own way through the structure of her program. I listened also to her doubts (expressed privately to me and to the others who sat in the office) that she would ever be able to make the work-around work, and of her fears of letting the others (who were relying on her finished code) down. I am particularly struck by Sima's continuous struggle with herself and her negotiation during her sleep of the routes the code would take and her effort to understand the way the code—if you like—flowed. Consider the following diary extract:

Sima has sat silent for several days now[,] only occasionally disturbed by Allison[,] who comes in periodically to check her progress. Sima asks her if she is worried that she will not get it done, and Allison says[,] “a bit.” Sima tells David and me how [the Department Manager] is scared to come and talk to her at the moment. I take it that this is because he has given her such a horrible job to do. The programmers are in many ways heroic figures. They are the “ones” who make things work and whom others rely on to do things.

Thus, one aspect of Sima's skill, then, is her ability to immerse herself in, and relate to, the code. However, to do so is about grasping the work of others (many others). This can often be a difficult thing to do. Finally, after a couple of weeks of struggling with the same piece of code, Sima relents and suggests to her manager that they should call in one of the Designers to help with the work-around.

Sima talked with the Designer (who was here for the day) about her problem of “making things work” and of how she is trying to change the code to print a “bank-check” instead of a “report.” They talk about details of the code. He sits beside her and suggests things to do. She has spent a lot of time on this. He tells her to try something, and he goes away to talk to Allison. Later he comes back to Sima, and finally they get the code to work. Their talk had been calm and “rational.” She was telling him what she had done, and he was suggesting to her what to try next.

Skill of this sort is neither a given nor an object, but has to be continually worked at and tested. To be at one with a technology, to use the code effectively, *takes effort*. How are we to understand the work of these “wizards”—in particular, *their* choice to carry out work-arounds? To speak of wizards is not to make a disparaging comment, for the Programmers that I observed were well qualified, highly skilled, and very motivated; rather, it is to emphasize the contingency and indeterminacy of work-arounds, and to suggest that the skill to perform them—to be in a position to make this *choice*—emerges from, and depends upon, networks elsewhere.¹⁵ Indeed, if you read some of

the recent literature on computer system implementation you will see that these difficulties are increasingly common. Georgina Born, in her study of the work of coding in a French research institute, writes about some of the problems of working on systems originally developed by others. The people she studied often complained that when they looked back on collaboratively written programs “. . . the complexity of the codes made it extremely difficult to reconstruct afterwards what was done, and how, in the bits of program authored by colleagues, without asking them” (1996, 109):

To manipulate the system effectively requires knowledge of the specific coded universe of different layers of code. Naive and inexperienced users are powerless to enter lower levels of the code hierarchy in order to alter or improve a program’s functioning. More surprising is the fact that the problem of the opacity of the hierarchy of codes—its resistance to meaningful decoding—also seriously affects senior . . . programmers (ibid., 109).

What is being suggested here is that rather than reduce everything to one simple determinant—for example, “it all comes down to skill,” we might think of skill as both a connection to certain networks and being able to perform the order embodied in those networks. This is, of course, the actor-network theory principle of treating actors as *effects*, and the view that technologies, among other things, have implications for us as agents (Law 1994). Thus a further aspect for understanding work-arounds is to consider how MAC itself provides for such modifications. Conventionally, we might think of MAC as a “passive” technology that is used by “active” agents who choose to use this tool in a number of different ways. Another way of imagining this would be to attempt to confuse this relationship between the Programmers and MAC. Actor-network theorists commonly speak of “hybrids”—that is, something different from just active humans and their passive technologies. It is to also emphasize that technologies are active, and that along with their users they “perform together” to produce “. . . the set of relations which give them their shape” (Law 2000, 5). Thus MAC, according to this way of thinking, is an actor in its own right. Moreover, if the skills of the Programmers are those of connecting and performing the order embodied in MAC, how do they perform together?

To explore this further I want to focus on a conversation I had with Maurice, another of the Programmers who worked in the office with David and Sima. Maurice characterized his experience of working with MAC in the following way: there is this constant need to make changes, as someone wants one part of the System to do something different, and he describes how MAC is “not built in concrete” and that “you *can* make changes to it.”

Maurice then goes on to acknowledge how the System also seems to work against his efforts to make changes:

I don't know if it was designed to be changed, however. Some of the code is tricky. I mean, it is doing some clever stuff. They must have some really clever people there, doing code better than I could do. Some of the code really takes a while for you to get your head around . . . The whole system is so constrained by the finance part of it. It is like a wheel with finance being the hub and the other parts being the spokes. You have to be careful when you make changes because you don't know what effect this will have on the other parts.

To clarify, Maurice seems to find himself in a position where the System is *asking* contrasting things of him: make changes/avoid making changes. It offers him the possibility of discretion in the sense that he is able to choose between different courses of action (Law 2000). MAC is not built in concrete and it can be changed. But, the way he decides to rewrite the code will affect others. For instance, changes he makes to the Finance part of the System will, among other things, affect the work of his colleagues who, elsewhere, are relying on his rewrites to allow them to get their own work done. MAC is central here because it can be easily modified, and it allows Maurice to decide on and attempt work-arounds. Indeed, numerous authors have commented on the abstract and malleable nature of software: Shapiro and Woolgar, for instance, make the argument that software *naturally* lends itself to “. . . all manner of personalized idiosyncratic development approaches” (1995, 16). They also make the point that for some programmers, they “. . . will primarily see opportunity while some will mainly feel burdened” by such malleability. The example of Sima unable to get her rewrite to work after a couple of weeks and being forced to call in a Developer, or Maurice's comment about having to be careful because of the Finance hub, are both illustrations of where the possibility of discretion is closed down.¹⁶ Here, MAC plays a part again because as it introduces its complex constraints—what Born (1996) earlier described as the “problem of the opacity of the hierarchy of codes”—there are very few possible courses of action.¹⁷

Conclusion

How do we account for a work-around? Often, the suggestion is that the user, when faced with a technology that is constraining in some form, is able to carry out a work-around and thus exercise some form of discretion or resistance.

This is always possible, especially if we understand the user and the technology to be each well bounded—that is, the role of the user is tightly defined as in a script, and the user attempts to work against this (also known as Akrich). However, if we consider new forms of computer systems and the prominent role the user is beginning to play in the shaping and customization of such systems, things are increasingly less clear-cut. MAC, like many of the computer systems increasingly used by organizations, is a flexible technology, or something of a “grey box,” in which users have the capacity to shape and customize the final design. What this suggests is that we will continue to witness more ambiguous sets of user-producer relations where it is often not clear who has responsibility for what. Because of these complex divisions of labor, various groups come to rely on each other as an integral part of their day-to-day working practices, often as resources for the resolution of technical difficulties and problems.

The actors discussed are not simply users but neither are they simply producers who have been attempting to routinely negotiate relationships and identities with others within these increasingly confused networks. Work-arounds represent one part of that negotiation process. And as we have seen with the MAC example, these connections are not simple or straightforward, but they are full of tensions. What I have hoped to achieve in this article is to convince the reader that there is arguably a need to develop an improved understanding of the practice and process of work-arounds in relation to these less determinate technical divisions of labor and responsibilities. Where this article adds to our understanding is through the description of some of the processes that might lead to work-arounds. In particular, as I have described, MAC and its associated networks provide not simply for one mode of use but, to paraphrase Michael (1996), they allow for multiple modes of use. Moreover, sometimes these contrasting modes will operate in unison and sometimes they will be in conflict. First, one aspect of this is that the Designers attempt to link the successful implementation of MAC to the Programmers and their ability to tailor the System to fit in with the existing software infrastructure. Following Latour (1992), I have described the competencies that the Designers appear to “bet on” as the “networks in place” of the Programmers. Thus, at one level, it would seem that the Programmers *actively* reconfigure MAC and the Designers enlist them in doing so. Secondly, there are some obvious problems with this. MAC itself *asks* for contrasting things from the Programmers. While MAC *can* be easily modified, and it allows Maurice to attempt work-arounds, it also introduces complex constraints (i.e., it acts against the possibility of work-arounds). A further element of the tension is that while work-arounds *are* demanded by the Designers, these practices are also sometimes proscribed, because as the

Programmers carry out their modifications, they call into question the Designers' responsibility toward MAC (i.e., the work-arounds infringe on the "80") and their role in the implementation. Hence, just *when* is a work-around a supported form of use, and when is it not, becomes a crucial question that has obvious resource implications, and this in itself makes it an important topic for the sociology of technology.¹⁸

NOTES

1. Such forms of use ranged from users entering inaccurate data to bypass weaknesses in existing systems, to users simply carrying out manually the procedures the computer system is meant to do and inputting the job *after* the work has been completed.

2. See the work of Claudio Ciborra (2002).

3. Kathryn Henderson (1999), for example, uses but does not develop the term in her recent book on engineers and their use of CAD. See also the article by Marc Berg (1997) where he describes how nurses work-around the limitations of a medical record system. For an example of how the notion of a work-around is used in the loose body of thought that comes under the heading of Computer Supported Co-operative Work (CSCW), see the article by Luff and Heath (1993). More recently, Button and Sharrock (1998) use the term to describe how programmers circumvent an "incompetent manager."

4. Scripts, argues Akrich, are often simply the outcome of decisions made by designers about future users—their skills and abilities and what the technology should do in relation to this user. Through the script: "... the designer expresses the scenario of the device in question—the script out of which the future history of the object will develop"(216).

5. See Berg (1997), who makes a similar point about writing within the social studies of technology.

6. This differs from Akrich, who describes a script as embedded within an artifact, whereas Michael is suggesting that scripts are both in the technology *and* in the wider networks attached to the technology. In other words, Michael's is a more dynamic notion of script where notions of use are the upshot of an interaction between the artifact and this larger network. A technology can hardly be thought as separate from, say, its instructions for use, as the artifact's working depends on these. In paraphrasing Pfaffenberger, he writes: "... technologies don't have instructions for their use inscribed in their design. Discourses are needed which guide users in their appropriate use"(1996, 3).

7. Friedman (1989), for instance, writing in the field of information systems, lists at least six user roles, which include not only those who simply input and retrieve data but also users who initiate systems and those who are involved in development and implementation as well as maintenance.

8. See the article by Button and Sharrock (1994) where they also describe programmers as users.

9. *Collins Concise Dictionary*, Fourth Edition, HarperCollins, 1999. For an explanation of this quote see the discussion below.

10. See Janet Rachel (1994), who makes a similar point when referring to her own ethnography and the apparent inactivity of the programmers she witnessed, though she notes that the *activity* of these programmers was "... produced through the appearance of inactivity" (819, *her emphasis*). Behind these seemingly still bodies, however, they were furiously typing away on

keyboards “. . . networked together in an effort to accomplish change on a grand scale in other parts of the organization”(819).

11. The apposite image of a true “Delphic” Support Desk is the one that I want you to keep in mind here.

12. See Ciborra (2002) for a detailed discussion of bricolage.

13. In their article, Star and Ruhleder (1996) ask *when* and not *what* is an infrastructure. Here, in their intriguing article, they are rehearsing the sociology of technology commonplace that technologies are not just things with particular properties “frozen in time” but emerge for people in the practice of technology use. Likewise, infrastructure, they argue, is also a fundamentally relational concept: “It becomes infrastructure in relation to organized practice. Within a given cultural context, the cook considers the water system a piece of working infrastructure integral to making dinner; for the city planner, it becomes a variable in a complex equation.”

14. The key paper for this form of ambivalence within the approach advocated by Akrich is Singleton and Michael (1996). They argue that while actor[MSOffice1]-network theory has tended to story “successful” networks as those where the actors strictly play-out their allotted roles, in practice actors often move between different positions (i.e., sometimes critical, sometimes supportive of the network). Indeed, as they argue, this crossover of roles often enables the very continuation of the network.

15. For a discussion of emergent skills, see Andrew Pickering’s *Mangle of Practice* (1995).

16. Leigh Star (1995) has described this as the “myth of infinite flexibility,” where in principle, software *can* be modified, but in practice it is very difficult to do so as changes will affect other parts of the system. This is especially true for integrated software systems (see Pollock and Cornford (2004) for a discussion of the difficulties of customizing Enterprise Resource Planning Systems).

17. For a discussion of software as a mediator, see also Born (1997).

18. The outcome of this negotiation will decide if the local programmers will receive further help in modifying that aspect of the system.

REFERENCES

- Akrich, M. 1992. The de-scription of technical objects. In *Shaping Technology/Building Society: Studies in Sociotechnical Change*, ed. W. Bijker, and J. Law. Cambridge, MA: MIT Press.
- Agre, P. 2000. Infrastructure and institutional change in the networked university. *Information, Communication & Society*, 4:494–507.
- Berg, M. 1997. Of forms, containers, and the electronic medical record: Some tools for a sociology of the formal. *Science, Technology, & Human Values* 22, 4:403–433.
- Berg, M. 1998. The politics of technology: On bringing social theory in to technological design. *Science, Technology, & Human Values* 23:456–490,468.
- Bijker, W. 1995 *Of Bicycles, Bakelites, and Bulbs: Towards a Theory of Sociotechnical Change*. Cambridge, MA: MIT Press.
- Born, G. 1996. (Im)materiality and sociality: The dynamics of intellectual property in a computer software research culture. *Social Anthropology* 4, 2:109.
- Born, G. 1997. Computer software as a medium: Textuality, orality and sociality in an artificial intelligence research culture. In *Rethinking Visual Anthropology*, eds. Banks, M. and H. Morphy. New Haven: Yale University Press.
- Brady, T., M. Tierney, and R. Williams. (1992). The commodification of industry applications software. *Industrial and Corporate Change* 1, 3:489–514.

- Button, G. and W. Sharrock. 1994. Occasioned practices in the work of software engineers. In *Requirements Engineering*, eds. Jirotko, M. and J. Goguen. London: Academic Press Ltd.
- Button, G. and W. Sharrock. 1998. The organization accountability of technological work. *Social Studies of Science* 28,1:73–102.
- Ciborra, C. 2002. *The Labyrinths of Information: Challenging the Wisdom of Systems*. Oxford: Oxford University Press.
- Friedman, A. 1989. *Computer Systems Development: History, Organisation and Implementation*. Chichester: John Wiley.
- Fujimura, J. 1992. Crafting Science: Standardised Packages, Boundary Objects, and “Translation.” In *Science as Practice and Culture*, ed. A. Pickering. Chicago: University of Chicago Press.
- Gasser, L. 1986. The integration of computing and routine work. *ACM Transactions on Office Information Systems* 4:257–70.
- Goddard, A., and P. Gayward. 1994. MAC and the Oracle family: Achievements and lessons learnt. *Axix* 1,1:45–50.
- Hales, M. 1995. Where are designers? Styles of design practice, objects of design and views of users in CSCW. In *Design Issues in CSCW*, eds. D. Rosenberg and C. Hutchinson. New York: Springer-Verlag.
- Henderson, K. 1999. *On Line and On Paper: Visual Representation, Visual Culture, and Computer Graphics in Design Engineering*. Cambridge, MA: MIT Press.
- Latour, B. 1992. Where are the missing masses? Sociology of a few mundane artifacts. In *Shaping Technology/Building Society: Studies in Sociotechnical Change*, eds. W. Bijker and J. Law. Cambridge, MA: MIT Press.
- Law, J. 1994. *Organizing Modernity*. Oxford: Blackwell.
- Law, J. 2000. “Economics as Interference” (draft), published by the Centre for Science Studies and Department of Sociology, Lancaster University at <http://www/comp.Lancaster.ac.uk/sociology/soc034jl.html>.
- Luff and Heath. 1993. System use and social organisation: Observations on human-computer interaction in an architectural practice. In *Technology in Working Order: Studies of Work, Interaction, and Technology*, ed. G. Button. London: Routledge.
- Michael, M. 1996. Technologies and tantrums: Hybrids out of control in the case of road rage. Paper presented at the “Signatures of Knowledge Societies” Joint 4S/EASST conference at the University of Bielefeld, Germany, October 10–13.
- Mol, A. and J. Messman. 1996. Neonatal food and the politics of theory: Some questions of method. *Social Studies of Science* 26,2:419–444.
- Newton, T. 1996. Agency and discourse: Recruiting consultants in a life insurance company. *Sociology* 30,4:717–739.
- Pfaffenberger, B. 1992. Technological dramas. *Science, Technology, & Human Values* 17,3:292–312.
- Pickering, A. 1995. *The mangle of practice: Time, agency & science*. Chicago: University of Chicago Press.
- Pollock, N. and J. Cornford. In press. ERP systems and the university as a “unique organisation.” *Information Technology & People* 17,1.
- Quintas, P., ed. 1993. *Social Dimensions of Systems Engineering: People, Processes, Policies and Software Development*. Hemel Hempstead: Ellis Horwood, 1993.
- Rachel, J. 1994. Acting and passing, actants and passants, action and passion. *American Behavioral Scientist* 37,6:809–823.
- Shapiro, D., and S. Woolgar. 1995. Balancing acts: Reconciling competing visions of the way software technologies work. Working paper, CRICT, Brunel University, UK, 16.

- Singleton, V. and M. Michael. 1996. Actor networks and ambivalence: General practitioners in the cervical screening programme. *Social Studies of Science* 23:227–264.
- Star, S. L., ed. 1995. *Ecologies of Knowledge: Work and Politics in Science and Technology*. New York: State University of New York Press.
- Star, S. L. and K. Ruhleder. 1996. Steps toward an ecology of infrastructure: design and access for large information spaces. *Information Systems Research* 7,1:111–134.
- Suchman, L. 1994. Working relations of technology production and use. *Computer Supported Cooperative Work (CSCW)* 2:21–39.

Neil Pollock is a lecturer at the University of Edinburgh, UK. His research interests include the sociology of software systems, virtual universities, and electronic commerce. His publications include Putting the University Online: Information, Technology & Organisational Change (with James Cornford, Open University Press, 2003). He is currently writing a book on the “biography” of software packages.