Organising User Interfaces Around Reflective Accounts

Paul Dourish[*], Annette Adler[†] and Brian Cantwell Smith[‡]

[*]Rank Xerox Research Centre, Cambridge Lab (EuroPARC)
[†]Systems Architecture Group, Xerox Corporation
[‡]Xerox Palo Alto Research Center

dourish@cambridge.rxrc.xerox.com, adler@parc.xerox.com, bcsmith@parc.xerox.com

Abstract

Over recent years, studies of human-computer interaction (HCI) from sociological and anthropological perspectives have offered radical new perspectives on how we use computer systems. These have given rise to new models of designing and studying interactive systems.

In this paper, we present a new proposal which looks not at the way in which we design systems, but at the nature of the systems we design. It presents the notion of an "account"—a reflective representation that an interactive system can offer of its own activity—and shows how it can be exploited within a framework oriented around sociologically-informed models of the contingent, improvised organisation of work. This work not only introduces a new model of interactive systems design, but also illustrates the use of reflective techniques and models to create theoretical bridges between the disciplines of system design and ethnomethodology.

1 Introduction

A spectre is haunting HCI; the spectre of ethnomethodology.

The past ten years have seen a significant change in the disciplinary constitution of Human-Computer Interaction and studies of interactional behaviour. What was once the domain of human factors and ergonomics specialists, and then became the domain of cognitive psychologists, has increasingly been colonised by sociologists and anthropologists. These disciplines have brought with them radically new perspectives on the way in which human-computer interaction is conducted.

One highly visible response to this shift in perspective has been the emergence of new ways of conducting interface design, and an increasing sensitivity to the details of users' everyday working practices. However, we will argue here that current work does not go far enough. Instead, what is required—if we are to take these new perspectives as seriously as they deserve—is a radical change, not only in the ways in which we build interactive systems, but in the nature of the systems thereby built. We will outline a new approach we have been developing based on the use of causally-connected reflective models of system and user interface behaviour. Reflection is, so far, the only computational model we have encountered with the power to address these issues. In this paper, we detail this new approach, and the elements of sociological analysis which underpin and motivate it; and we argue that this represents not only a significant new move in HCI, but also a critical area for the development of reflective principles in everyday systems.

1.1 Ethnomethodology and HCI

In this paper, we will draw upon a particular branch of sociology called ethnomethodology [Garfinkel, 1967]. A growing number of researchers have been using this and allied approaches in the analysis and design of interactive systems.

Ethnomethodology's roots lie in a respecification of the issues of sociology. In particular, it reacts against a view of human behaviour that places social action *within* the frame of social groupings and relationships which is the domain of traditional sociological theory and discourse—categories and their attendant social functions. Ethnomethodology's primary claim is that individuals do *not*, in their day to day behaviour, act according to the rules and relationships which sociological theorising lays down. Quite the opposite. The structures, regularities and patterns of action and behaviour which sociology identifies emerge *out of* the ordinary, every-

Dourish's current address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304. dourish@parc.xerox.com.

Adler's current address: Architecture and Document Services Technology Center, Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304. adler@parc.xerox.com

Smith's current address: Department of Computer Science, Lindley Hall, Indiana University, Bloomington, IN 47405. bcsmith@cs.indiana.edu

day action of individuals, working according to their own common-sense understandings of the way the social world works. These common-sense understandings are every bit as valid as those of learned professors of traditional sociology.

From this basic observation, a new picture of social action has arisen. It would neither be possible nor fruitful to detail it here, but some simple characterisations are critical. The ethnomethodological view emphasises the way in which social action is not achieved through the execution of pre-conceived plans or models of behaviour, but instead is *improvised* moment-to-moment, according to the particulars of the situation. The sequential structure of behaviour is *locally organised*, and is *situated* in the context of particular settings and times.

Ethnomethodology's concern, since its beginning, has been the organisation of human action and interaction. In 1987, Suchman published *"Plans and Situated Actions,"* which applied the same techniques and perspectives to the organisation of interaction between humans and technology. In doing so, she opened up significant new areas of investigation both for HCI researchers and ethnomethodologists. The same techniques have now been applied to a wide range of settings within HCI research, and most particularly have become a significant component in research on Computer-Supported Cooperative Work (CSCW). This perspective has lent weight to the analyses and critiques of interactive technology, lending weight to the emergent Participatory Design movement, and similar approaches encouraging new models of HCI design practice.

Our focus, though, is at a more fundamental level. We want to explore the implications of this new sociological view not just for the *ways in which we build* the artifacts of HCI, but for the *nature of the artifacts themselves*; and, from there, we want to understand what new artifacts would look like that take the ethnomethodological perspective seriously.

1.2 Traditional Process HCI

The traditional view of interface work is strongly *process-based*. From this perspective, the function of the interface is to guide the user through the regularised, well-understood sequence of actions by which some goal is reached. The process is uncovered (or made visible) by requirements analysis, and subsequently encoded (or made invisible) in design.

This traditional, process-oriented view structures the way in which interfaces are designed, evaluated and studied. Indeed, the regularisation it embodies extends to interface design methodologies and formalisms, and can be seen, for example, in the use of formal "automata" structures in the description of interface activity, from interface transition diagrams to workflow graphs and business process models.

The alternative view, which arises from sociological investigations such as those cited above, is at odds with this process orientation. Instead, it focuses on work as the improvised management of local contingencies, and emphasises the way in which regularisations in the conduct of activity at the interface arise out of individual moment-to-moment action (rather than the other way around). In this view, work is not so much "performed" as *achieved* through improvisation and local decision-making.

It is on this tension that we have been focussing attention. Since computational design is inherently prescriptive, and, of necessity, involves abstractions of action and processes, how can it be made responsive to this view of the improvised and unfolding organisation of user behaviour?

1.3 Improvisation and Resources

One starting point might be to ask, "how does the 'process' of improvisation proceed?" Unfortunately, this would be the wrong question, and one that is almost impossible to answer. Ethnographers spend years detailing the particular ways in which particular activities are organised. What we can do, though, is step back from the detailed descriptions and try to draw out some general issues from the broad sweep of these investigations.

Our starting point, then, is the view that the actions which constitute work at the interface are locally organised; indeed, the work process as a whole emerges from this sequence of locally improvised actions. So the question which is going to concern us is, what kind of information goes into this local organising process? How are the improvisational "decisions" made? If our goal is to support this character of work, then a critical focus of design must be to provide information or *resources* which support and inform the local, expedient decision-making process, rather than to formalise and encode the process itself. And once we have some idea what the information might be, we're then in a position to ask, "how can it be applied?"

2 Operation and State

When humans use computer systems to perform some work, it is clear that an important resource in the improvised accomplishment of their activity is their belief about the *state* of the system. What is it doing? How much has it done? What will it do next? Why did it do that?

These questions are based on system state, and shape the sequential organisation of action. It's important to realise, though, that it's not the state information itself that's valuable. After all, a user is generally engaged in accomplishing some other work with the system, rather than performing a detailed study of its behaviour. The information about what the system is doing is only useful when it helps the user's task proceed (or helps the user to progress). So what's really important to the user is the *relationship* between the state of the system and the state of the work which the user is trying to accomplish. When such a relationship can be established, then information about the state of the system can be used to understand and organise on-going working activity.

Relevant state information is readily apparent in most devices we deal with day to day. Wheels turn, bits of paper come in and out, and curious noises (and sometimes smells) emerge. Visual access, operating noises and observable behaviour all provide information about the system's state from moment to moment; we can see and hear information about the state of devices and mechanisms which we might want to use. In saying that, though, it's important to recognise the distinction between what we see and hear, and what we *understand* about device state. On their own, the various resources accessible to the user aren't of much use; some greater context is needed before they become meaningful.

In particular, a user's view of the relationship between the state of the system and the state of their work is rooted in some belief—however incomplete, inaccurate or naive—about *how the system works*. This is true for almost anything. For instance, the variety of resources which support the activity of driving a car—such as the sound of the engine and the feel of the clutch—only make sense within the context of some (possibly incorrect) model of how a car works in the course of getting someone from A to B. This lets us interpret not only the information, but also its consequences for our activity. Similarly, activity (particularly "situated" activity) is organised around, and depends upon, these sorts of understandings; they allow us to ask questions like, "what is the system doing?", "what do I want to do next?", and "how should I go about it?".

The next question is, where do these understandings come from? Clearly, there are many sources. One of the most important is our own everyday experiences, and the pictures of structure and causation which we build up as a result of daily interactions with all sorts of devices. Other sources include the experiences of others, in stories, advice and anecdotes; others include formal instruction—courses, manuals, and so forth. One other important source is essentially *cultural*—the everyday understandings of devices which we gain as a result of living in a world of Euclidean space, Newtonian mechanics and the internal combustion engine.

Clearly, however, one of the most important components of this understanding is the *story the system tells about itself*—how it presents its own operation and state (and the relationship between the two). Some of this is explicit, being part of the way in which we might interact with a device; some may be more implicit, such as the noises which devices make in operation. Explicit or implicit, though, it all contributes to the story.

Mechanical (or partly mechanical) devices are physically embodied, and right there in the world with us, giving us this information. Indeed, when a photocopier needs a paper jam cleared out, we might get more information than we bargained for. Gibson's [1979] notion of the *affordances* for action which a situation offers to appropriately-equipped individuals begins to relate activity to this notion of embodiedness. Software systems have no observable physical embodiment, though, and so the user interface is the only place where the user can get a view into the system's machinery.

Aspects of the interface and the way it behaves are suggestive of the system's capabilities, and of the sorts of temporal or causal constraints acting on it. These contribute to the understanding the user builds up of the system's operation and the relationship of its components and activity to the work the user is attempting to perform. Again, this presentation has both explicit aspects (*e.g.* the iconic representations in direct manipulation interfaces) and implicit ones (*e.g.* the dynamic or temporal properties of interactions).

So if we want to support improvised action, then we have to focus on two things—on the resources, presented in the interface, that support the improvisation; and, critically, on the model the system presents of its own behaviour, which contributes to the context in which these resources can be interpreted and hence supports improvisation.

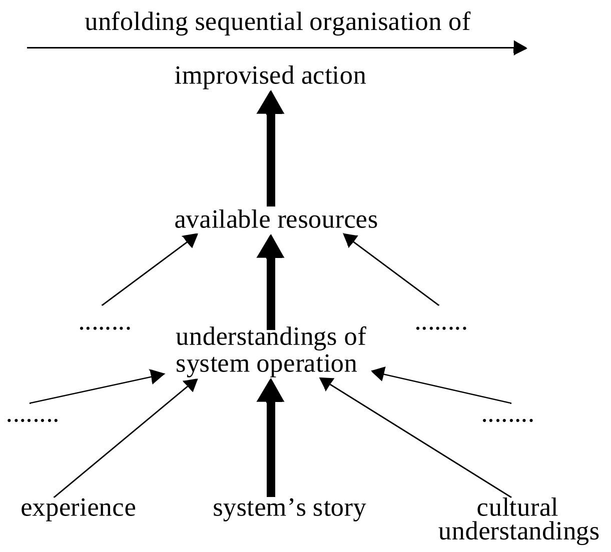

FIGURE 1: The resources that underpin improvised action are interpreted in a context which is formed, in part, by the story the system tells about its own behaviour.

There's clearly meta-ness here; the system is representing itself, and so there's clearly a leaning towards a reflective solution. But before going further, we must consider how the type of information we're considering here is dealt with in existing systems; and that will turn out to be familiar to the reflection community, too.

3 Connection and Disconnection

Given the importance of this “self-revealing” aspect of the interface, we must ask what the relationship is between the presentation that the interface offers and the *actual* operation of the system. How is this relationship structured, and how is it maintained? These are important questions, and they lead us to identify a problem in maintaining this relationship—a problem of *connection*.

Before worrying about how information about system activity should be presented to the user, we need to understand how the interface component can find out what’s going on in the first place. There are essentially two ways that an interface can discover information about the activity of underlying system components. The first is that it may be constructed with a built-in “understanding” of the way in which the underlying components operate. Since the interface software is constructed with information about the semantics and structure of the other system components to which it provides a user interface, it can accurately present a view of their operation. In view of the strong connection between the application and interface, we’ll call this the *connected* strategy. The second, *disconnected* strategy is perhaps more common in modern, “open” systems. In this approach, the interface component has little understanding of the workings of other system components, which may actually have been created later than the interface itself, and so it must infer aspects of application behaviour from lower-level information (network and disk activity, memory usage, *etc.*). Essentially, it *interprets* this information according to some set of conventions about application structure and behaviour; perhaps the conventions that support a particular interface metaphor.

However, there are serious problems with both of these approaches. The connected approach is the more accurate, since it gives interfaces direct access to the structure of underlying components and applications. However, this accuracy is bought at the expense of cleanliness and modularity. This is clearly bad practice; but perhaps, if it were the only problem, it would just be the price we have to pay for effective interface design. Unfortunately, it’s not the only problem. Perhaps more critically, *extensibility* is also broken. Because of the complex relationship between interface and application, a new application cannot be added later once the interface structure is in place. The interface and application cannot be designed in isolation, and so a new application cannot be added without changing the internals of the interface software. The result is that this solution is inappropriate for generic interfaces, toolboxes and libraries, which provide standard interface functionality to a range of applications.

So what of the disconnected approach? The problem here is that, while it leads to modular and extensible designs, it is not reliably accurate. The relationship between the low-level information it uses and the higher-level inferences it makes is complex and imprecise. Also, there are problems of synchronisation. Because the representations of activity that the disconnected approach manipulates are implicit, its inferences can be consistent with the available information but out-of-step with the actual behaviour of the system. This approach, then, is largely heuristic; and so its accuracy cannot be relied upon, particularly for detailed information.

Essentially, the connected approach is *too* connected, and the disconnected approach is *too* disconnected.

3.1 Example: Duplex Copying

As a way of grounding this problem, imagine a digital photocopier. It offers various familiar system services—such as copying, scanning, printing, faxing—as well as other computationally-based functions, such as image analysis, storage/retrieval and so forth. A generic user interface system provides the means to control these various services, perhaps remotely over a network.

Somebody wishes to use the copying service to copy a paper document. The paper document is 20 pages long, printed double-sided (*i.e.* 10 sheets), and the user requests 6 double-sided (“duplex”) copies. Half way through the job, the copier runs out of paper and halts.

What state is the machine in? How many copies has it completed? Has it made 3 complete copies of the document, or has it made 6 half-copies? The answer isn’t clear; in fact, since copiers work in different ways, it could well be either. However, the critical question here concerns the interface, not the copier *per se*. How does the interface component react to this situation? What does it tell the user is the state of the device? And, given that this is a generic interface component which was constructed separately from the copier’s other services, how does the interface component even *know* what to tell the user, or how to find out the state?

This situation doesn’t simply arise from “exceptional” cases, such as empty paper trays, paper jams and the like. It also occurs at any point at which the user has to make an *informed judgement* about what to do next, such as whether to interrupt the job to allow someone else to use the machine urgently, whether it’s worth stopping to adjust copy quality, and so forth. Even the decision to go and use a different copier requires an assessment of the current machine’s behaviour. What these situations have in common with the exception case of an empty paper tray is that, as users, we must rely on the interface to support and inform our action, even when we find ourselves stepping outside the routinised “process” which the interface embodies. When the interface presents system activity purely in terms of the routine—or when its connection to the underlying system service gives it no more information than that—then we encounter the familiar tension between technological rigidity and human flexibility.

4 Accounting for System Action

The elements of the story we have presented so far resonate strongly with ideas which the reflection community has explored since the early 1980's. The problems of self-representation and disclosure in section 2 are essentially the same as those tackled by 3-Lisp [Smith, 1982]; and the problems of connection and abstraction barriers in section 3 are essentially those of Open Implementation [Kiczales, 1992; 1996]. It seems natural, then, that we should look towards the principles and techniques of computational reflection for solutions to the problems we have set out, and for the foundation of a new form of interactive system design.

Just as open implementations address problems of connection between system components, we can use the same approach to address the "interface connection" problems of section 3. So consider an alternative view of an open implementation's reflective self-representation. Consider it as an "account" that a system component presents of its own activity. Being a self-representation, it is generated from within the component, rather than being imposed or inferred from outside; being reflective, it not only reliably describes the state of the system at any given point, but is also a means to affect that state and control the system's behaviour.

Such an account has a number of important properties. It is an *explicit* representation—that is, computationally extant and manipulable within the system. It is, crucially, *part of the system*, rather than simply being a story we might tell about the system from outside, or a view we might impose on its actions. It is a *behavioural* model, rather than simply a structural one; that is, it tells us how the system acts, dealing with issues of causality, connection and temporal relationships, rather than just how the system's elements are statically related to each other. However, the account itself has structure, based on defined patterns of (behavioural) relationships between the components of the account (perhaps relationships such as *precedes*, *controls*, *invokes*, and so forth).

Most importantly, we place this requirement on the account—that it "*accounts for*" the externally-observable states of the system which presents it. That is, it is a means by which to make a system's behaviour accountable. The behavioural description which the system provides should be able to explain how an externally-observable state came about. This critical feature has various implications, which will be discussed shortly. First, however, let's return to the duplex photocopying example.

4.1 Accounting for Duplex Copying

If we adopt this notion of "accounts," then the copy service (which provides copying functionality in the copier, and which lies below the interface component alongside other system services) provides not only a set of entry points to its functionality—the traditional abstraction interface, often called an "Application Programming Interface" or API—but also a meta-interface or account, a structured description of its own behaviour. The API describes "what the service can do"; the account describes "how the service goes about doing it". It describes, at some level, the sequence of actions which the service will perform—or, more accurately, a sequence of actions which *accounts for the externally-observable states of the system*. So, if the interface has access to details not only of the functionality offered by the copying service, but also an account of how it operates in terms of page copying sequences and paper movement, then it can provide a user with appropriate information on the state of the service as it acts, and continuation or recovery procedures should it fail.

So, this notion of reflective self-representations as "accounts" provides a solution to the problems raised in the duplex copying example. More importantly, in doing so, it also provides a solution to the connection problem raised in section 3. The interface module does not have to *infer* activity information (as was necessary with the disconnected interface strategy). Instead, it can present information about the system accurately because the information it presents comes directly from the underlying components themselves (where it is causally connected to their actual behaviour). At the same time, information about the structure and semantics of those components is not tacitly encoded in the interface module (as it was in the connected interface strategy). Instead, this information is explicitly made available from the components themselves. It is manifested in accounts they offer of their actions which the interface module can use, preserving the modularity and extensibility properties of a disconnected implementation. This balance between the connected and disconnected approaches is maintained through the two critical aspects of the reflective approach: *explicit representations* and *causal connection.*

To understand the ways in which accounts can support interface activity, we first have to look in more detail at the properties of accounts themselves.

5 Exploring Accounts

Accounts and reflective self-representations are essentially the same thing; our use of the term "accounts" connotes a particular perspective on their value and use. By the same token, the familiar properties of reflective representations also apply to accounts; but they may have particular consequences for a use-oriented view.

One important issue, which derives from our grounding in research on Open Implementations, is that accounts reveal aspects of *inherent structure* rather than the details of specific implementations. In other words, the account presents a rationalised model of the behaviour of the system, revealing some details and hiding others, as required by the purposes to which its designer intends it to be put. It both enables and constrains. The account stands in a two-way semantic relationship to the implementation itself; this much is guaranteed by the causal connection. But that relationship is not a direct one-to-one mapping between the elements of the implementation and the elements of the account. We can perhaps think of an account as being a particular *registration* of the implementation; a view of the implementation which reveals certain aspects, hides others, and highlights and emphasises particular relationships for some specific purpose.

So the account need not be "true" in an absolute sense; it is accurate or precise for the purposes of some specific use, in context. The system may well have to go to some lengths to maintain the validity of the account in particular circumstances. Imagine, for instance, that the "copying" account of section 4.1 presented, for simplification, a model in which only one page was being processed at any moment. However, even fairly simple copiers typically process multiple sheets concurrently, to increase throughput. This would be perfectly valid *as long as* for any observable intermediate state—that is, any point where a user (or user interface) might intervene in the process, either through choice or necessity—the system can put itself into a state which is accounted for purely in terms of the model offered.

Naturally, this begs the question: what states are observable? There is no absolute answer to this question; like any other reflective representation, not only does it depend to some extent on the structure of implementations, but it also depends on the *needs* of the user in some particular situation. This reflects a tension in the account between *accuracy* and *precision*. The account must, at all times, be accurate; that is, in its own terms, it must correctly represent the state and behaviour of the system. However, this accuracy may be achieved by relaxing its precision, the level of detail which the account provides. Relaxing precision allows the system more flexibility in the way it operates.

The invariant property, though, is that of *accountability*; that the system be able to account for its actions in terms of the account, or that it should be able to offer an account which is not incompatible with previously offered accounts. In these terms, accountability is essentially a form of *constructed consistency*. This aspect of the account draws further on the relationship between account-oriented improvisation of activity and the ethnomethodological perspective presented earlier. Accounts and representations in social interaction are given their authority and validity by the pattern of social relationships which back them up, and by which one is, to a greater or lesser extent, held *accountable* to one's words and actions. So, the utility of an interface account depends on the backing that the system offers—in this case, the guarantees sustained by the causal connection. It is this notion of accountability, based in the direct relationship between action and representation, which is at the heart of this proposal, and which distinguishes accounts from simple simulations.

However, accountability is by no means the only significant property deserving discussion here. Another cluster of issues revolve around accounts being inherently *partial*. An account selectively presents and hides particular aspects of a system and its implementation. It is *crafted* for specific purposes and uses. By implication, then, it is also *variable*; the level of detail and structure is dependent on particular circumstances and needs, as well as the state of the system itself at the time.

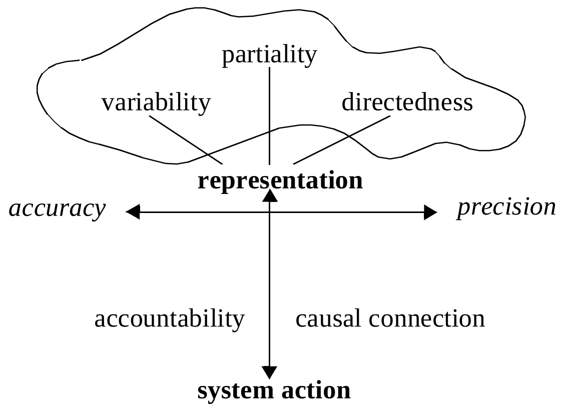

FIGURE 2: The account lives in a balance between accuracy and precision. When precision is loosened, through partiality, etc, the causal connection sustains its accountability.

This is another area where the balance between accuracy and precision becomes significant. This variability must also depend on the recipient of the account, which is *directed* towards specific other entities, be they system components or users. The whole range of ways in which accounts are only partially complete and are designed for particular circumstances (in a way that reflects the balance of needs between the producer and receiver of the account) is reflected in the use of the term "account". Included in this is the principle that variability is dynamic; the account is the means by which structure and information can be gradually revealed, according to circumstances. To draw again on the ethnomethodological metaphor this variablility corresponds to the idea of *recipient design* in conversation analysis; the crafting of specific utterances for a particular recipient or audience. This level of specificity also emphasises that accounts are *available for exploration*, rather than being the primary interface to a system component. We don't have to deal in terms of the account at all times, but we can make appeal to it in order to understand, rationalise or explain other behaviour.

One final property is important here. Again as derived from reflective self-representations, an account is *causally connected* to the behaviour it describes. It is not simply "offered up" as a disconnected "story" about the system's action, but

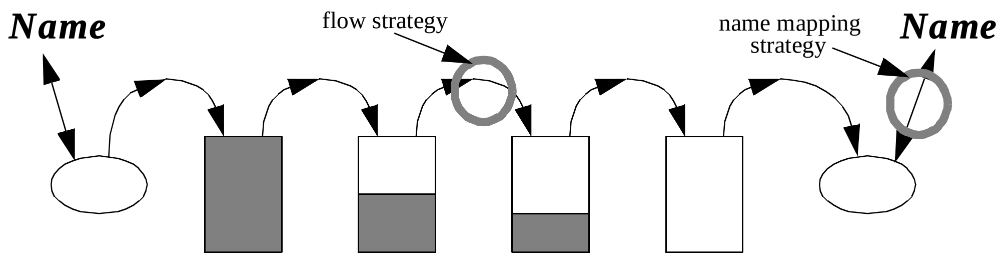

FIGURE 3: A structural model of the file copying example in terms of data buckets and the connections between them. Connections between elements of this model are the points at which strategies and policies can be identified.

stands in a more or less connected causal relationship to it. Changes in the system are reflected in changes in the representation, and vice versa. The critical consequence of this is that the account be computationally *effective*—an account provides the means not only to describe behaviour, but also to control it. The link between the account and the activity is bidirectional. The account is a means to make modifications to the way in which the system works—it provides the terms and the structure in which such modifications are described. Indeed, the structure of the account clearly constrains the sorts of modifications that are allowed, whether these are changes to the action of the system itself, or—more commonly, perhaps— manipulations of the internal processing of specific jobs in progress.

6 Accounts and Users

Previous sections used an example of a duplex copying task as an illustration of the value of an account-based approach to system architecture. The copying example illustrates one way of using these representations. The use of accounts in that example is derived fairly directly from explorations over the past few years of the use of reflective representations and metalevel architectures in system design. At the system level, reflective representations or accounts can provide a critical channel of communication between system components or modules, and in particular offer a solution to the problem of connection in generic interfaces.

However, it is interesting to examine a more radical use of accounts—their use at the *user level*. The goal here is to address more directly the disparity that was highlighted in the introduction, between the improvised, resource-based nature of actual work and the process-driven model assured in classical interface design. The accounts model is an attempt to address this by thinking of computational representations as resources for action. On the one hand, the account mechanism builds directly on the importance of the "stories systems tell" about their activity; and on the other, the causal connection and principle of accountability (or constructed consistency) supports the variability of use. Accounts provide a computational basis for artful action.

6.1 Example: File Copying

Let's consider a second example—a real-world interface problem with its origins in a breakdown of abstraction. Imagine copying a file between two volumes (say, two disks) under a graphical file system interface. You specify the name of the file to be copied and the name of the destination file; after you start the copy, a "percentage done" indicator (PDI) appears to show you how much of the copy has been completed. This generally works pretty well, especially when the two volumes are both connected to your own machine. But consider another case, which isn't so uncommon. You want to copy a file from a local volume to a remote volume on a nearby fileserver over a network. This time, when you copy the file, the PDI appears and fills up to 40% before the system fails, saying "remote volume unavailable". What's happened? Was 40% of the file copied? Did all of the file get 40% there? Most likely, none of the file ever reached the remote volume; instead, 40% of it was *read* on the local disk before the machine ever tried to reach the remote volume. What's more, there's no way to tell *how* the remote volume is unavailable; on some systems, this might even mean you don't have your network cable plugged in (and so the remote volume was *never* available). Finally, a failure like this makes you wonder... just what's the PDI telling you when things *are* working?

In general, there's simply no way to see at which point in the copy failure occurred, since the interface presents no notion of the structure or breakdown of behaviour and functionality that's involved. In fact, the notion of a partially-completed copy makes little sense when offered in the interface, since the interface doesn't even offer terms in which to think about what's going on. What does it mean when the copy is partially completed, and when the PDI indicates there's more to do?

We can begin to address this problem by looking for the inherent structure of the example. Start by reifying the various areas where data might reside at any moment; files, buffers, caches, the network, interface cards, etc. The details are not important; they're specific elements of an implementation, rather than inherent features. The essential point is simply that there are some number of these "data buckets"; that some are files and some are not; and that the process of copying a file involves connecting a series of them together to get data from one place to another. So we end up with a structure rather like that in figure 3.

In this figure, we see a set of data buckets connected together, indicating the flow of data between two points. Some of these buckets (the end points) are files; they exist independently of the particular copy operation, and are distinguished with names[1]. The other data buckets are temporary intermediate ones. The flow of data through the system is determined by the strategies used at the connection points between the data buckets. A wide range of mechanisms could be used: flushing a buffer on overflow or an explicit flush, transferring data between buffers in different units, etc. The point isn't which mechanism is used in any given case. Rather, it is that the account gives the interface—and the user—a structure and vocabulary for describing the situation. In terms of this vocabulary, aspects of system behaviour can be explicated and controlled.

So when the particular configuration in some given situation is available for exploration, we can begin to answer questions about the interface and system behaviour. Just as the set of flow strategies characterises the flow of data through the system as a whole, so the flow can be controlled through the selection of strategies; and the behaviour of the percentage-done indictor is connected to (characterised and controlled by) the point in this sequence where it is "attached". Should it be attached towards the left-hand side, for instance, then it will tend to reflect only the local processing of data—not its transmission across the network, which is often of greater importance to the user, and which caused the failure in the case we were considering[2]. However, without any terms of reference, it isn't possible to talk about "where" the indicator is attached—far less to move it around. When needed, then, the account provides these terms of reference; an explicit structure within which specific actions can be explained, and their consequences explored. This structure—one within which exploration and improvisation can be supported—is not supported by traditional interactive software structures which make details inaccessible behind abstraction barriers.

This account is aimed at solving interface problems arriving from the traditional file system abstraction, which arise because file system operations are characterised purely in terms of *read* and *write* operations. This takes no account of whether the operations are performed locally or remotely, and the consequences of such features for the way in which the interface should behave. The abstraction has hidden the details from higher levels of the system, but those details turn out to be crucial to our interactions[3].

1. In fact, naming is a separate issue in the account which a system provides; in this example, its relevance is that the source point named is a file, whereas the end point is given a name *before* a file exists there. However, the issue of naming is not discussed in this example.

2. Note a second extremely confusing—and potentially dangerous—failure which can result here. The PDI can indicate 100% copied, before the remote volume complains that it's full after writing only 40% of the file. Which report should be trusted?

This example illustrates a number of general points on the nature and use of accounts. First, consider the relationship between the model and the system itself. Unlike other approaches to interface visualisation, the model arises from the structure of the system and is *embodied in* the system. It is not imposed from outside. It is general, in that it does not reflect the details of a particular implementation, but rather reflects the inherent structure of all (or a range of) implementations. It is a gloss for the implementation, explicitly revealing and hiding certain features deemed "relevant".

Second, consider the relationship between the account and the activity. The causal relationship renders the account "true" for external observation; because it is of the system itself, rather than simply of an interface or other external component, it is reliable in its relationship to the actual behaviour represented. However, the level of detail it presents reflects the balance between accuracy and precision; while it accurately accounts for the behaviour of the system, it only reveals as much as is necessary for some particular purpose—in this case, explaining the curious "40% complete then 100% failure" behaviour.

Third, it allows us to talk not only about structure, but about "strategies"; that is, it is a behavioural model, not simply an architectural one. This means that the system can break down and "reason about" policy and strategy. An account is not simply a name for a way of doing something, but describes the pattern of relationship between its constituent activities; and this is critical to the way it's used.

7 Perspectives and Conclusions

There is a tension between the traditional process-oriented view of user interfaces and interaction—interfaces as currently designed—and the view of interface work as the locally-improvised management of contingencies that has been emerging over the past ten years or so. This tension becomes particularly troublesome when we attempt to incorporate some of the insights of sociological investigations into system design. In this paper, we have argued that addressing this problem not only means rethinking the way in which we go about systems design, but also leads to a new approach to the nature of the systems which we design. In focusing on the resources that support improvised work at

3. In fact, problems of this sort can be seen in a wide range of systems where network filestores have been grafted on within the abstractions designed for local filestores, because "you needn't worry if the file is local or remote".

the interface, we have been concerned here with how users understand system activity, and in particular with the way that systems and devices find and present such information. This reveals a problem in the structure of interactive systems—a problem of *connection* between system components.

Accounts are causally-connected representations of system action that systems can offer as explications of their own activity. They are inherently partial and variable, selectively highlighting and hiding aspects of the inherent structure of the systems they represent, but, being views of the system from within rather than without, they are reliable representations of ongoing activity. A system is held accountable to its account; that is, the account must adequately “explain” the observable states of the system that offered it.

This work is part of an ongoing investigation of the relationship between sociological and ethnomethodological perspectives on work and interaction and the practice of systems architecture and design. A number of groups, particularly investigating the use of collaborative technologies, have attempted to integrate ethnomethodology into their design methods. The approach we have been exploring, however, addresses this integration as a theoretical, as well as a practical, concern [Button and Dourish, 1996]. In our work over the last two years, we have focussed on the use of reflection and metalevel implementation techniques to address problems in system architecture and use. The explication and reification of semantic structures in the reflective approach, making them amenable to examination and manipulation, has provided an opportunity to focus not on how usage issues can be encoded within systems, but rather, at how the flexibility inherent in everyday activity can be, itself, the subject of computation. Rather than attempting to “lift the system to the user’s level” (for instance, through the use of AI techniques), or “lower the user to the system’s level” (by forcing users to address their work in system terms), we have been exploring, instead, how the mediation between these two levels can be flexibly and fruitfully accomplished.

Acknowledgments

Many colleagues have contributed to these ideas and to this presentation of them. We are indebted to Bob Anderson, Graham Button, Beki Grinter, Austin Henderson, Gregor Kiczales, David Levy, Gene McDaniel, Bob Printis and Randy Trigg for comments, inspiration and support in the development of this work.

References

[Button and Dourish, 1996] Graham Button and Paul Dourish, “*Technomethodology: Paradoxes and Possibilities*”, Proc. ACM Conference on Human Factors in Computing Systems CHI’96, Vancouver, Canada, May 1996.

[Garfinkel, 1967] Harold Garfinkel, *“Studies in Ethnomethodology”*, Prentice-Hall, New York, 1967.

[Gibson, 1979] J. J. Gibson, *“The Ecological Approach to Visual Perception”*, Houghton Mifflin, New York, 1979.

[Kizcales, 1992] Gregor Kiczales, “*Towards a New Model of Abstraction in the Engineering of Software*”, Proc. IMSA Workshop on Reflection and Metalevel Architectures, Tokyo, Japan, November 1992.

[Kiczales, 1996] Gregor Kiczales, “*Open Implementations*”, IEEE Software, pp. 6—11, January 1996.

[Smith, 1982] Brian Smith, “*Reflection and Semantics in a Procedural Language*”, MIT Laboratory for Computer Science Report MIT-TR-272, 1982.

[Suchman, 1987] Lucy Suchman, *“Plans and Situated Actions: The problem of human-machine communication”*, Cambridge University Press, Cambridge, UK, 1987.