Accounting for System Behaviour: Representation, Reflection and Resourceful Action

Paul Dourish

Rank Xerox Research Centre, Cambridge Lab (EuroPARC)
and Department of Computer Science, University College, London

dourish@europarc.xerox.com

Abstract

A clear tension exists between the traditional process-oriented view of interface design and the emerging improvisation-oriented view of interface activity, which arises particularly from sociological investigations of computer-based work. This paper attempts to address this disparity and begin to bridge the gap, focussing on technological foundations and implications, in a way which makes the insights of sociological investigations “real” in computational design.

It presents a novel approach to interface architectures, based on the use of explicit, causally-connected self-representations in computational systems. These are treated as “accounts” which systems offer of their own activity. The paper traces some of the consequences of this approach both for system design and interaction, and shows how it addresses current problems in the design of flexible interactive systems.

1 Introduction

Over the past ten years or so, the performance of computer-based work has increasingly become a focus of investigation for social scientists. Studies such as those of Suchman [1987] or Heath and Luff [1992] examine work-at-the-interface with the same sorts of techniques and orientations which are applied to other forms of work, looking at the wider social and organisational settings in which such work takes place, and the methods by which work is organised moment-to-moment. We might expect that studies from a sociological or anthropological perspective would complement those from more psychological or system-oriented standpoints. Instead, however, a tension has arisen between the general tenor of their conclusions and more traditional views of interfaces and interface design.

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. This traditional 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 Statecharts to workflow graphs.

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.

Clearly, there is a tension between these two perspectives; and this tension becomes particularly prominent when we attempt to relate the findings (or even simply the tone) of sociological investigations to the needs of interface design. In this paper, my concern is to address this disparity and to discuss a model of design which sits more comfortably with these investigations of improvised work. Our starting point, then, is the contrast between process and improvisation.

1.1 Improvisation and Resources

In trying to understand how a view of improvised work relates to the needs of system design, we might start by trying to consider the “process” of improvisation itself. How does this process proceed, and how is it managed? Unfortunately, these questions are unwieldy and ill-formed. Long-term ethnographic investigations of system use seek to find detailed answers in very specific situation so we could scarcely expect to set down general answers here. However, we can 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. If our goal is to support this character of work, then a critical focus of design must be the provision of the information or *resources* which support and inform the decision-making process, rather than the formalisation and encoding of the process itself.

This notion of design around resources reflects a concern not simply with the *how* we go about system design—such as is a central concern of Participatory Design, for example—but with the *artifacts* of that design process. It suggests a change in the structure of computational systems we build, and later I will introduce a model of computation which provides support for this form of design. The principle of explicit design around resources for local decision-making has been observed elsewhere (e.g. [Dourish and Bellotti, 1992]; [Dourish, 1993]; [Robinson, 1993]), but here I want to explore it more explicitly, and follow through the consequences for the interactive systems we design.

It would be foolish to imagine that the design process can capture *all* the relevant information which informs and supports the improvisation of computer-based work. Even for this most restricted domain, this would be not only impossible, but wrong-headed. The factors underlying such situated activity are unbounded, and derive not only from the system, but also from knowledge of social and organisational situations, from ongoing interactions with others, and so forth. Rather, what is proposed here is a model of design which reveals some information which may be of value; and which is sensitive to the improvised character of use. So there are various reasonable questions we *can* ask. What kinds of information can systems provide which may be of value in supporting improvisation in various contexts? What does it mean to take this as a focus of system design? How deep a change to our model of systems is implied?

I will begin by looking at important factors which shape how people interact with systems, and in particular, how systems present models of their operation. I will then go on to consider the status of such models in existing systems, and how we can exploit them to bridge between the conclusions of use analysis and the needs of system design.

2 Operation and State

When an individual uses a computer system to achieve 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? These questions are based on system state, and shape the sequential organisation of action, but the state information itself is not intrinsically valuable. A user is generally engaged in accomplishing some other work with the system, rather than performing a detailed study of its behaviour; and 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 which we deal with day to day. 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 use aren't of much use; some other 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 the activity is rooted in some belief—however incomplete, inaccurate or naive—about *how the system works*. This is true for all devices which mediate or support our activity. 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—are useful only within the context of some model of how a car works, or at least simple understandings such as that of the relationship between engine pitch and the rev count. Such an understanding allows a user (or driver) to interpret not only the information, but also its consequences for their activity. By the same token, activity (particularly "situated" activity) is organised around, and depends upon, these sorts of understandings; they allow us to mediate between questions like, "what is the system doing?", "what do I want to do next?", and "how should I go about it?".

Where do such understandings come from? Obviously, there are many sources; and so everybody's understandings are highly personalised. One of the most important sources is our own everyday experiences, and the general pictures of structure and causality which we build up as a result of daily interactions with all sorts of devices. Other sources include the experiences of others, related through stories, advice and anecdotes; whole others again come from more formal learning and instruction procedures—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*—its presentation of its own operation and state (and the relationship between the two). Certain aspects of such a presentation are explicit, being part of the way in which we might interact with a device; others may be more implicit, such as the noises which devices make in operation. Explicit or implicit, though, they all contribute to the story.

However, computational "devices" (software systems) tend not to be physically embodied—a printer is embodied, but a network filesystem is not[1]. So the presentation of this information—the story that the system tells about itself—arises primarily at the user interface[2]. 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. Along with all the other factors cited above, these contribute to the understanding the user builds up of the systems'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).

Although the implications are much broader, the work discussed in this paper is focussed very specifically on this issue—how a system presents a story about its actions. In particular, I will be concerned with the way in which the interface represents and conveys this information. Traditional interfaces may work hard to convey useful information about ongoing action; however, since traditional interface design is so heavily organised around the "process" model, this doesn't address the problem of directly supporting the improvised nature of working activity. Here, then, I will focus particularly on the implications of telling a richer story at the interface, and doing so precisely to move towards a model of design motivated primarily by situated accounts of everyday activity.

Before considering how systems might embody and manipulate such "stories", I will first discuss how they arise in traditional systems.

3 Connection and Disconnection

Given the importance of this aspect of the interface, we must ask, what is the relationship between the presentation 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.*

The issue here is not how information about system activity should be presented to the user, but rather, 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 the 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, I'll call this the *connected* strategy. The second, *disconnected* strategy is perhaps more common in modern, "open" systems. In this approach, the interface 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 each 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 modularity and extensibility. Modularity is broken because there are now a variety of complex semantic relationships between details of the interface behaviour and details of the application behaviour. The two are linked by a web of complex interconnections, making it very difficult to change one without necessitating changing the other. This is regarded as 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 simply 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 is modular and extensible, 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 which 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 it's accuracy cannot be relied upon, particularly for detailed information.

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

1. Or rather, the embodiment is generally not a part of the user's world.

2. Note the familiar corollary—that any aspect of the system which reveals information *is* part of the interface.

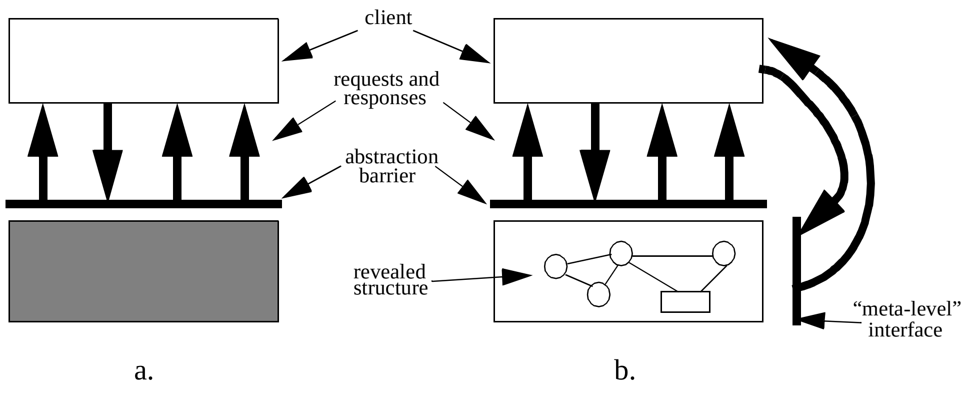

FIGURE 1: (a) Clients interaction with traditional black-box abstractions through standard abstraction barriers. (b) Open implementations also reveal inherent structure.

3.1 Example: Duplex Copying

Let's pause to consider an example 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 is 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 Open Implementations

A potential solution to this general problem is to be found in another area of computer science where similar problems with the mappings between high-level descriptions and lower-level actions have arisen. *Open implementations* [Kiczales, 1992] arose originally as an approach to the design of programming languages, balancing elegance and descriptive power in the language against efficiency over a range of specific implementations. More generally, however, it has been applied to a range of areas of system architecture and design which share a range of common problems, focussed around the issue of *abstraction.*

Traditionally, the notion of abstraction, as derived from mathematics, has been one of the most powerful tools in the creation of software systems. The "black box" abstraction (as illustrated in figure 1a) *encapsulates* certain structures and behaviours and makes them available to other system components through an *abstraction barrier*. The barrier isolates the *implementation* of the abstraction from the components which use it (its *clients*). This separation serves two functions. First, it makes it possible to reuse software components without having to understand all the details of their construction. Since the developer of the client software need only understand how to use the abstraction which the software presents in order to make use of it, much larger and more complex software systems can be built by combining and reusing component modules. Abstraction aids the *manageability* of large problems. Second, it allows the client software to be independent of the details of the abstraction's implementation. Since the client refers only to the elements of the abstraction itself, rather than to elements of the implementation, any implementation which accords to the same abstraction could be used. So, as long as well-defined abstractions are the only point of contact between software modules, new pieces of software can be substituted for old ones, and so systems can be maintained and improved in a reliable, modular way. Abstraction gives us a way of defining and managing *equivalence*.

This model of abstraction is based on one fundamental principle: that, as long as the abstraction offered by a software module is accurately maintained, the details of how that abstraction is implemented *do not matter* to other system components. Indeed, by this principle, they *should* be hidden, to encourage independence between modules. However, this assumption is becoming increasingly open to question. The problem is that the abstractions we use in software engineering are not the pure abstractions of mathematics. Instead, they are simply the *visible aspects of underlying implementations*; and, like icebergs floating in the sea, the structure of what lies hidden below is just as critical to how we should treat them as the visible aspects above. The structure of the implementation has important consequences for the way in which clients will use the abstraction. What's more, aspects of the implementation will always show through the abstraction barrier, revealing themselves in size limitations, performance limitations and dynamic properties of the module.

The open implementation approach is shown in figure 1b. Here, a system provides not only a standard interface to the abstract structures and behaviours offered by this module, but also a separate interface to the *inherent structure* of that module. This interface allows clients to view aspects of the structure (*introspection*) but also, crucially, to make modifications (*intercession*); to "reach in" to the implementation and adapt it to the needs of specific clients. It provides access to a computational "meta-level", which talks not about what the system does, but about how it does it. This approach has been fruitfully applied to a range of areas of system development, including the design of the Common Lisp Object System [Bobrow *et al*, 1988] and the Silica window system [Rao, 1991]. A full treatment of the technical aspects and consequences of this approach is too long to present here—the interested reader is referred to Kiczales *et al.* [1991]. However, certain aspects are worth discussion here.

The essence of this approach is *computational reflection* [Smith, 1982]—the notion that a system maintains and has access to a representation of its own behaviour. The crucial element of such reflective representations is that they are *causally connected* to the behaviour they describe. Thus, changes in the system are reflected in changes in the representation; and, critically, the behaviour of the system can be changed by making changes to the representation.

At the same time as we introduce these ideas, it is important to retain various important properties of the existing notion of abstraction, principally the conceptual simplification which it provides (the manageability principle). There are two ways in which this is achieved in an open implementation. First, a standard or default interface is provided; that is, the interface to the meta-level representation *augments* the traditional abstraction barrier, rather than replacing it. Second, the view into the implementation reveals its *inherent* structure, rather than the details of a specific implementation. It does not simply provide a set of "hooks" directly into the implementation; that would both constrain the implementor of the abstraction and require too much of the implementor of the client. Instead, it provides a rationalised model of the inherent behaviour of a system offering its particular functionality. Clearly, this view of what is "inherent" is a normative one; typically, it is conditioned by an understanding of the specific failings of standard abstractions in some particular domain. I will return to this issue later. For the moment, though, I will consider how this revised view of abstraction can be used to solve the sorts of problems in interface structure which were raised earlier.

5 Accounting for System Action

Just as open implementations address problems of connection between system components, we can use the same approach to address the "interface connection" problems related earlier. So consider an alternative view of an open implementation's reflective self-representation. Consider it as an "account" which a system or module presents of its own activity. A self-representation, it is generated from within the component, rather than being imposed or inferred from outside; 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 structural properties, 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. 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.

5.1 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 an 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.

6 Exploring Accounts

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

One important issue raised in the description of open implementations was that they reveal aspects of *inherent structure* rather than the details of specific implementations. Just as that was critical to retaining the conceptual simplification and separation which is so valuable in abstraction, it is also critical to the value of accounts. After all, there is little benefit in separating two system components if each is forced to understand every detail of the other to use it. Instead, we exploit the system's inherent structure and present a rationalised view of the behaviour of a module. The inherent structure captures elements such as the relationship between the components presented at the abstraction barrier; aspects which are implied by the existence of those elements and their roles in the abstract behaviour described. 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*[3] 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. Indeed, it is possible that the system will 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 5.1 presented, for simplification, a model of operation in which only one page was being processed at any moment. However, during normal operation, the system might actually work on several pages at once—indeed, even fairly simple copiers typically do this in order 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; not only does it depend on the structure of an implementation, 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

3. My use of this term, and the flexibility it implies, draws on Brian Smith's treatment of representation, and especially computational representation. [Smith, *forthcoming*].

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*. 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 simply simulations.

However, accountability is by no means the only significant property which deserves discussion here. Another set of properties revolve around the fact that accounts are inherently *partial*. An account selectively presents and hides particular aspects of a system and its implementation. Indeed, the account 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. 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 which 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 further on the metaphoric structure of this proposal, these properties can be thought of as embodying properties similar to those of the term *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.

There is one final property which 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 stands in a 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 is 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 which are allowed, whether these are changes to the action of the system itself, or—more commonly, perhaps—are manipulations of the internal processing of specific jobs in progress.

7 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. At the system level, 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. This use of accounts is derived directly from the general perspective of open implementations, and the use of causally-connected self-representations to address failures in the notion of "abstraction" in software engineering. As such, this use of accounts focuses on issues in the architecture of interactive systems.

However, it's 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 which was highlighted in the introduction, between the improvised, resource-based nature of working and the process-driven aspect of 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.

7.1 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; you start the copy and 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 PDB 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.

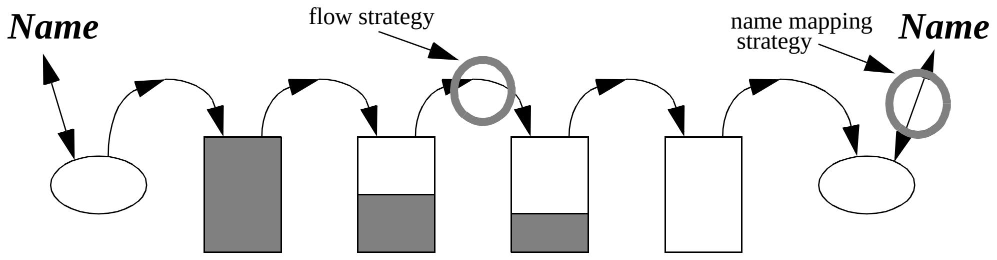

FIGURE 2: A structural model of the file copying example in terms of data buckets and the connections between them. Connections between elements of this model are the points at which strategies and policies can be identified.

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 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. There are various places where data can reside—let's call these *data buckets*. Some of them, perhaps, are files; others may be areas of temporary storage. The network itself, for instance, is a data bucket but not a file. In addition, there are caches, network interface buffers, and so on. 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 2.

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[4]. 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. There are a wide range of mechanisms which could be used:

- the *flush on overflow* strategy. A bucket accumulates data until it's full; at that point, all the data in that bucket is "flushed" into the next bucket;
- the *trickle on overflow* strategy. A bucket accumulates data until it's full; at that point, new incoming data displaces equal amounts of old data into the next bucket;
- the *chunking on overflow* strategy. A bucket accumulates data until it's full; at that point, a fixed amount of data (a "chunk") is flushed to the next buffer to make space;
- the *explicit flush* strategy. A bucket accumulates data until it is explicitly told to flush it all to the next bucket.

This is by no means intended to be an exhaustive list. Rather, it is intended to illustrate the wide range of strategies which could be used. It can be seen that the choice of strategies at each point—and there may be different strategies at each point—characterises the flow of data from one end to the other.

We can map this inherent structure onto various specific situations, such as the case of networked file copying. There is some number of data buckets, corresponding to the various relevant entities in the system. Entities might include the files themselves, the filesystem cache, the network interface, the network itself, and so on. The precise set of elements involved is not directly important; the inherent structure, and its relation to the implementation, is of much greater significance, and the existence of a bucket is often more important than its identity. 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 process-

4. 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.

ing 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[5]. 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.

The basic problem reflected in this example arises directly from the traditional view of abstraction discussed in section 4—in this case, the use of abstraction inside the file system. It arises 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[6].

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. The model arises from the structure of the system; but it is also *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, consider the importance of explicit references to "strategies". Strategies—normally implicit in the creation of a piece of software—are *reified* or rendered explicit in this model. This accords to a general principle in the engineering of large, flexible software systems, the separation of *mechanism* (the means for accomplishing action) and *policy* (the means of deciding what action is appropriate). However, it also extends this model, since policies are given as structured behavioural models. This means that the system can break down and "reason about" a policy. 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.

5. Note a second extremely confusing—and potentially dangerous—failure which can result here. The PDB 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?

6. 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".

8 Perspectives and Current Work

This paper has used some simple examples to show the way in which accounts can be used to solve problems in system interaction. In particular, these problems have arisen because traditional notions of representation in computational systems provide poor support for the management of contingencies that arise in the course of conducting some activity. Whether we look at the interactions of people with systems or of system components with each other, we find problems which arise from the way in which conventional models of abstraction break down.

It's important to recognise that a reflective representation is still a representation; it is still a normative description of some system. An account is a designed artifact, and it incorporates a set of assumptions and expectations about usage patterns in exactly the same way as other artifacts do. The key difference, though, is in the way in which the representation is related to the system or activity it represents. In a conventional system, not only is the representation of system action partial, but so is the way in which the representation and action are connected. In effect, this constrains not only the information which the representation can reveal, but also the ways in which the representation can be used.

The account mechanism addresses this relationship between representations and the behaviours they represent, and opens up the way in which these representations can be used. This is one step on a path towards a model of computational design which is rooted firmly in studies of the nature of working activity, and in attempting to understand their implications not only for the way in which design is done, but also for the nature of the artifacts which are designed. The reflective approach is also being employed in the design of Prospero, a toolkit for CSCW applications [Dourish, 1995].

Ongoing activities focus on other issues arising from this model of representation and interaction. For instance, nothing has been said here of the issues surrounding where an account originates. Like the structure of the account, these might also have their origins in understandings of how users approach, use and understand systems. In the case of photocopying, we should perhaps look to a notion of "naive xerography" as providing a starting point for our description of how systems operate. The case of hybrid electronic/mechanical systems, such as photocopiers, is particularly interesting because of their embodied nature. Photocopiers make noises, produce output, and occasionally exhort users to open them up and muck around inside; a computational account is no use at all if it is directly belied by the clear and obvious path of paper through a machine.

It's critical, though, to recognise and maintain the distinction between accounts and mental models of device behaviour—the kinds of understandings of the world which people bring to bear when interacting with systems and devices. Clearly, the two are strongly related. However, they exist on different sides of the interface. We distinguish between an *account of system behaviour* as offered by a system, and the *understanding of system behaviour* formed by a user in response. Accounts are explicit technological artifacts—computational representations which stand in special semantic relationships to the systems they describe.

9 Summary 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 to contingencies which has been emerging particularly 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. I have argued that addressing this problem means not only rethinking the way in which we go about systems design, but also a new approach to the nature of the systems which we design. In taking a focus on the resources which support improvised work at the interface, I 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 which systems 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 which offered it.

This is an attempt, then, to look at the balance in interface design between *abstraction* and *detail*; on the one hand, the abstraction and generalisation which is inherent in the process of software construction, and on the other hand, the detailed moment-to-moment activity which makes up the work at the interface. The hope is that accounts, as described here, provide a means by which users can more accurately match the functionality of a system or device to the immediate requirements of the practical accomplishment of their work; and more generally that they point the way towards a deeper relationship between the insights of observational analysis and the practice of systems design.

Acknowledgments

Many colleagues have contributed to these ideas and to this presentation of them. I am particularly indebted to Annette Adler and Austin Henderson, who helped me start thinking about these issues in the first place, and to Bob Anderson, Graham Button, Beki Grinter, Brian Smith and Randy Trigg for comments, inspiration and help in the development of this paper.

References

[Bobrow *et al*, 1988] Daniel Bobrow, Linda Demichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales and David Moon, "*Common Lisp Object System Specification*", X3J13 Document 88-002R, June 1988.

[Dourish and Bellotti, 1992] Paul Dourish and Victoria Bellotti, "*Awareness and Coordination in Shared Workspaces*", Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '92, Toronto, Canada, November 1992.

[Dourish, 1993] Paul Dourish, *"Culture and Control in a Media Space"*, Proc. Third European Conference on Computer-Supported Cooperative Work ECSCW93, Milano, Italy, September 1993.

[Dourish, 1995] Paul Dourish, *"Developing a Reflective Model of Collaborative Systems"*, ACM Transactions on Computer-Human Interaction, 2(1), 40–63, March 1995.

[Heath and Luff, 1992] Christian Heath and Paul Luff, *"System Use and Social Organisation"*, in Button (ed.), "Technology in Working Order: Studies of Work, Interaction and Technology", Routledge, 1992.

[Kizcales *et al*, 1991] Gregor Kiczales, Jim des Rivières and Daniel Bobrow, *"The Art of the Metaobject Protocol*", MIT Press, Cambridge, Mass., 1991.

[Kiczales, 1992] Gregor Kiczales, "*Towards a New Model of Abstaction in Software Engineering*", Proc. IMSA'92 Workshop on Reflection and Metalevel Architectures, Tokyo, Nov 4–7, 1992.

[Rao, 1991] Ramana Rao, *"Implementational Reflection in Silica"*, Proc. European Conference on Object-Oriented Programming ECOOP'91, Geneva, Switzerland, July 1991.

[Robinson, 1993] Mike Robinson, *"Design for Unanticipated Use"*, Proc. Third European Conference on Computer-Supported Cooperative Work ECSCW93, Milano, Italy, September 1993.

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

[Smith, *forthcoming*] Brian Smith, *"On the Origin of Objects"*, MIT Press, Cambridge, Mass., *forthcoming.*

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