PUMA Programmable User Modelling Applications

Programmable User Modelling Analysis for usability evaluation

Ann Blandford

Jason Good & Richard M. Young

WP11a
April 1998

Principal Investigator	Dr Ann Blandford	Middlesex University
Research Fellow	Dr. Richard Butterworth	Middlesex University
Academic Collaborator	Dr David Duke	University of York
Research Fellow	Jason Good	Middlesex University
Industrial Collaborator	Sue Milner	Praxis Critical Systems Ltd
Academic Collaborator	Dr Richard Young	University of Hertfordshire

http://www.cs.mdx.ac.uk/puma/

Contact : Dr Ann Blandford
School of Computing Science
Middlesex University
Bounds Green Road
London. N11 2NQ. UK
tel : +44 (0)181 362 6163
fax : +44 (0)181 362 6411
email : a.blandford@mdx.ac.uk

Project funded by
EPSRC grant number GR/L00391

Section 1: Introduction to Programmable User Modelling in HCI

Section 2: Programmable User Modelling in practice

0. Introduction

The aim of this tutorial is to provide a brief general introduction to cognitive modelling as it relates to work on Human-Computer Interaction, and to give you the basic skills in applying one particular style of modelling, Programmable User Modelling Analysis (PUM Analysis, or PUMA). So that the material can be presented or worked through in discrete units (each of which is intended to take 3-4 hours in a lecture or tutorial context), these notes are split into two sections. The intention is that each section should be self-contained, though section 2 presumes knowledge of section 1. Thus, it is possible to just work through section 1 and (hopefully) go away with a good general understanding of what PUMA is "about" as an approach to cognitive modelling. Section 2 builds on that understanding, and relates it more to large-scale practical projects.

By the end of this tutorial, you should have a reasonable understanding of:

a) what a cognitive model is (and isn't);

b) what cognitive modelling techniques are good for, and in particular how to identify situations that PUMA is suited to; and

c) the basics of the cognitive theory that underpins PUM Analysis.

Feedback and suggestions for improvements are most welcome.

Section 1: Introduction to Programmable User Modelling in HCI

1. Introduction to cognitive modelling in HCI

We start with a general overview of cognitive modelling as applied within HCI — focusing on techniques that allow us to make predictions about user performance when working with devices. This first chapter provides a general introduction to cognitive modelling in HCI. Subsequent chapters in this section will give an introduction to Programmable User Modelling and an overview of PUM Analysis, including small case studies.

What is a "cognitive model"?

A "model" is a representation of something that is constructed for a purpose. So, for example, a flight simulator is a model of an aircraft cockpit that enables pilots to learn how to fly a particular type of aircraft, and deal with emergency situations in safety, but it does not enable anyone to travel from Heathrow to Boston. Similarly, a cognitive model is a representation of ways people think that is developed for a purpose. For cognitive psychologists, the main purpose is often to give them a framework for talking about the way people think; in HCI, the purpose is more often to give designers a way of thinking about usability issues from a user-centred perspective.

Cognition is concerned with how people think — i.e. with rational, reasonable behaviour. So, for example, it may cover how people solve problems, learn, remember, or perceive things. It does not, in general, cover non-rational aspects of human behaviour such as falling in love, feeling happy or depressed, or having a phobia. In terms of use of computers, of course, this means that we can talk about a high proportion of the relevant concerns.

A cognitive model, then, is a representation of the way a person (the user of a computer system) thinks.

Cognitive models and cognitive architectures

Some cognitive models are based explicitly on cognitive architectures. An architecture is a structure that constrains how the system behaves. So, for example, the layout of rooms in a house or an office building influences the way that building is used; how strong the influence is depends on how general or specialised the architecture is. Similarly with cognitive architectures — though in this case, the architecture underlying a model determines how that model behaves, not how real people do! A cognitive architecture is a theory, or a framework, for understanding cognition, and cognitive modelling involves working with the representations or constraints imposed by the architecture being used.

While some cognitive architectures, such as ACT-R[1] and Soar[2], have been implemented in computer systems, very little cognitive modelling work in HCI makes direct use of such architectures. It takes too much time, effort and expertise to construct such detailed models. So their main contribution is to developing a better understanding of the features a usable cognitive modelling technique needs to have.

PUMA is based on symbolic modelling; i.e. it includes an explicit representation of knowledge or

---

[1] See ANDERSON, J. R. (1993). *Rules of the Mind*, Hillsdale, NJ: LEA.

[2] See NEWELL, A. (1990) *Unified Theories of Cognition*, Harvard University Press, Cambridge, MA.

beliefs and an explicit mechanism that determines how knowledge is used to determine behaviour.

Most work on symbolic cognitive modelling can be traced back to that of Allan Newell and others[3] on the General Problem Solver (GPS) and the Model Human Processor (MHP). This early work was based on experimental results of people being asked to perform certain kinds of problem solving (such as syllogistic reasoning — e.g. "All farmers rise at dawn; Brenda rises at dawn; is it true that Brenda is a farmer?" or "All bakers get up at 3am; Bill is a baker; is it true that Bill gets up at 3am?"). It focused very much on mental tasks that took little account of the role of the environment in problem solving, and on problem solving of the "right or wrong" variety.

More recently, the essentially interactive nature of work with a computer system has been acknowledged, and interactions now tend to be modelled in terms of user and device taking turns, and the user using the device as a resource.

What is cognitive modelling?

A simple definition of cognitive modelling is that it is the process of constructing a model of the way a person thinks, and then reasoning from that model. Some models are paper-based descriptions that can be used to focus the analyst's thinking and to support reasoning about behaviour. Others are "simulation models" that can be run to simulate particular aspects of the way we think and behave. Such models are implemented as computer programs, and produce output that can be compared with people's behaviour; this comparison can be a way of assessing the validity of the model.

A lot of cognitive modelling is based on the notional goal of constructing a computer simulation, but without "going that far". The focus of such modelling work is generally on producing an appropriate representation of the knowledge that the target model would need if it were to be built and run. But here we're getting into too much detail too early....

Why do cognitive modelling in HCI?

There are two main reasons for doing cognitive modelling in Human-Computer Interaction:

> a) to predict what aspects of a particular device might be easy or difficult to use and/or learn. To anticipate areas where people are likely to make persistent errors.
>
> b) to develop a better general understanding of how people interact with computer systems, and what makes computer systems easy or difficult to use.

The natural assumption, for most people, is that the main point of cognitive modelling is to have a cognitive model to look at, observe the behaviour of, or show to other people. Although this may be important in some cases, in many instances it is a secondary concern; the main insights in terms of design come from the process of building the model, rather than from the end result[4]. Producing a cognitive model forces the analyst to describe the design in a particular way, to make assumptions explicit and to face up to aspects of the design that are contradictory, inconsistent, or under-specified. It is only when a complete and consistent model has been produced that it can possibly be run, to find out what behaviours are predicted. But most of the value of modelling can be obtained earlier in the process.

Approaches to cognitive modelling in HCI

To illustrate various approaches to cognitive modelling, let us consider a non-HCI task, and

---

[3] See NEWELL, A. AND SIMON, H. (1972). *Human Problem Solving,* Englewood Cliffs, NJ: Prentice Hall.
or: CARD, S. K., MORAN, T. P. AND NEWELL, A. (1983). *The Psychology of Human Computer Interaction,* Hillsdale : Lawrence Erlbaum.

[4] This same argument has been made for the use of formal methods in design — that the main insights come from the process of describing the problem in a particular, rigorous, way rather than from looking at the product (a formal specification).

discuss in general terms how it would be analysed using three techniques: GOMS[5], Cognitive Walkthrough[6] and PUMA. The task we'll use for this is doing your grocery shopping.

A GOMS analysis involves describing the task structure and decisions made by the user in terms of Goals, Operators, Methods and Selection rules. A goal is something the user wants to achieve; in the case of grocery shopping, this might be expressed as "User possesses items A, B, C…". Operators are the low-level actions the user can perform (such as picking an item off the shelf, or signing a credit card slip; the analyst chooses an appropriate level of detail according to the style of analysis being conducted). Methods are the procedures needed (i.e. the sequence of operators to be applied) to achieve goals; for example if we defined a "paying by credit card" method, it might consist of the steps:

open wallet;
take out credit card;
hand card to cashier;
sign credit card slip;
retrieve card, slip and receipt;
return card to wallet.

We could leave this as the lowest level of analysis, or we could break some of these steps down still further, creating a hierarchy of methods. This is not the only way of paying (i.e. of achieving the goal of having paid for the goods); for example, the analyst might also define methods for paying by cash or cheque. If there are multiple methods that address the same goal, the analyst should also define Selection rules to specify under what circumstances the modelled user would select each of the methods.

For something as routine as paying, it is easy to define methods. If it were a useful thing to do, the user's behaviour could be completely modelled for any given set of initial conditions, giving approximate times to complete the overall task.

For the larger task of doing the shopping, it is less easy to define methods — unless the shopping is a completely routine activity, such that one could say that it consists of getting a loaf of brown bread, then getting 2 pints of semi-skimmed milk, then getting half a kilo of Cheddar cheese, and so on. Using GOMS, it is hard to describe the opportunistic nature of most real shopping activity, for which a more realistic main goal might be "User possesses appropriate food for the week", and for which real user actions might include picking up a new kind of dessert because it looks interesting, or choosing 4 cans of brand X beans, rather than 1 of brand Y because X is on special offer this week.

GOMS is well suited for analysing routine tasks, for which the user knows all the relevant information about the system they are working with and which can be described in terms of procedures.

Cognitive Walkthrough is aimed more at analysing exploratory behaviour, in which the user doesn't know in advance how to achieve their goals, but learns it through the interaction. Cognitive Walkthrough is normally conducted by several analysts, who agree a task scenario (e.g. the Bloggses want to buy all their food for the week, including holding a dinner party for 8, at which they plan to serve risotto and a range of desserts), a user profile (the Bloggses do most of their shopping at the SaveALot superstore, so they are most familiar with that style of shop), an action sequence for achieving the task, and a description of the interface. Each analyst then "walks through" the action sequence, aiming to tell a "success story" (or, failing that, a failure story) about how the user will achieve the intended effect and know that the effect has been achieved. Cognitive Walkthrough focuses primarily on how easy it is to learn to use a new interface (or, in this case, where things are in an unfamiliar shop) rather than how easy it is to subsequently use

---

[5] See CARD, S. K., MORAN, T. P. AND NEWELL, A. (1983). *The Psychology of Human Computer Interaction*, Hillsdale : Lawrence Erlbaum.
Or: JOHN, B. & KIERAS, D. E. (1996). Using GOMS for user interface design and evaluation: which technique? *ACM ToCHI*. **3.** 287-319.

[6] See WHARTON, C., RIEMAN, J., LEWIS, C. AND POLSON, P. (1994) The Cognitive Walkthrough method: a practitioner's guide. In J. NIELSEN AND R. MACK, Eds. *Usability Inspection Methods*. pp.105-140. Wiley : New York.

that interface. So a Cognitive Walkthrough analysis of shopping would concern itself with the signposting and labelling around the shop (If the user doesn't know where the frozen peas are, will they be able to find this out? Will the user easily know where to pay? Will the user be able to quickly discover whether or not the shop sells Maraschino cherries??). The task of paying for the goods, which is easy to describe using GOMS, is much more difficult to describe using Cognitive Walkthrough, because it is not the kind of task that is usually learned through exploration.

PUMA has much in common with each of these approaches, but focuses more on how the user deploys knowledge in achieving their goals. The analyst describes the objects the user is working with — such as foodstuffs, locations and cash, relationships between those objects (e.g. how much cash each product costs, where in the store a product is displayed, what products are in the user's trolley), and operations that the user knows about. In this case, the most important operational knowledge the user needs to have is that to buy a product it must first be in the trolley, that to put it in the trolley it must be on a nearby shelf (visible to the user), and that the way to find a product is to search the shelves and walk around, following signs if there are relevant ones around. If the user is near a product that is on the list (so that the "precondition" for putting it in the trolley is satisfied) the user can put it in the trolley. The sources of knowledge available to the user from the environment consist of labels and his or her ability to recognise a product from its packaging or appearance. In addition, the user may use knowledge from prior experience — for example, that lemon juice is classified as a cooking ingredient, not a fruit juice. By laying out the knowledge in this kind of way, PUMA can be less prescriptive about the precise procedures a user will follow to achieve goals, and focus more on the knowledge the user needs, and how that knowledge is made available to them. While it would be difficult to use PUMA to describe all the opportunistic decisions most shoppers make, it does allow the analyst to be less prescriptive about procedures than other task-oriented techniques, and to account for some of the ways in which the design of the environment influences behaviour. The knowledge needed for paying would be rather more complex than that laid out for GOMS — e.g. that to have paid, the user must have signed the credit slip, that a precondition for this is that the user has handed the credit card over to the cashier, which in turn necessitates having the card in the hand, etc..

While cognitive modelling techniques have much in common, each is particularly well suited to analysis of situations with particular characteristics. The model that underlies the Cognitive Walkthrough technique focuses on novice users who are using a device in an exploratory way; GOMS considers mainly routine expert behaviour; PUMA is particularly concerned with users' knowledge and the ways knowledge is used in the interaction.

2 PUMA in design

The philosophy behind the Programmable User Model (PUM) approach to cognitive modelling is that an analyst (a member of the design team) should have an "empty" user model — i.e. a fixed architecture — that they program with domain- and device-specific knowledge. They also have to produce a device description (Figure 1) so that they can investigate the interaction between the user and device .

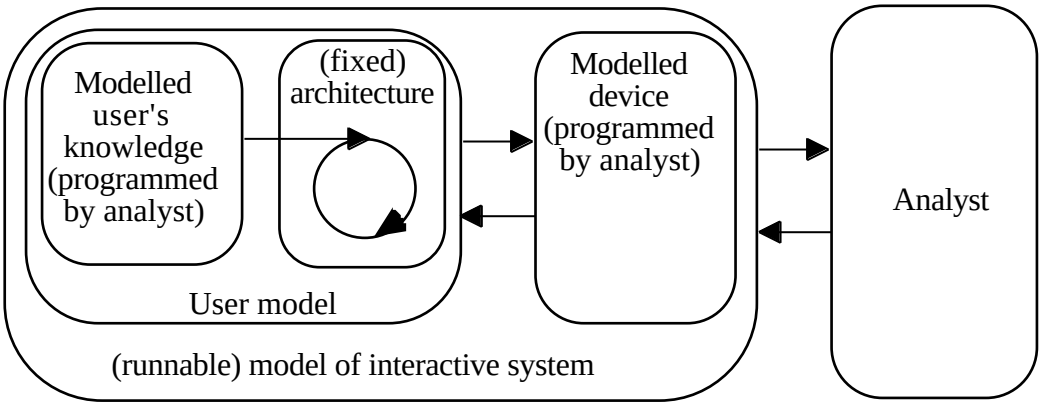


Figure 1: the role of the analyst

The analyst develops a better understanding of user concerns in the design task both by going through the process of specifying the user's knowledge in the required format, and by observing the behaviour of the running model (see figure 2). The language used for specifying user knowledge is called the Instruction Language (IL). In practice, in this tutorial, we focus mainly on describing the user's knowledge, and less on running models.

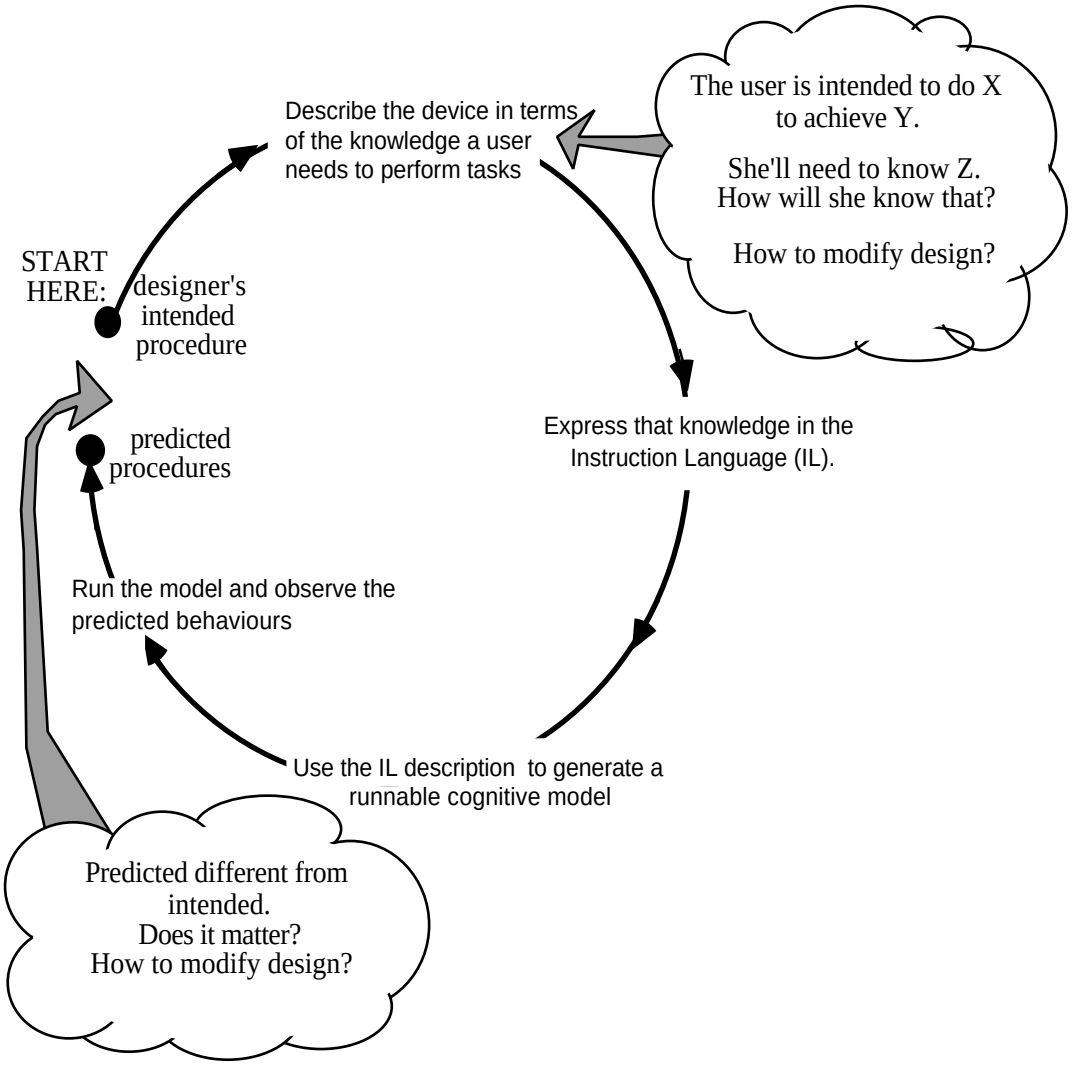


Figure 2: PUM in the design process

Different design problems "succumb" to analysis in different ways. Some can be dealt with very superficially: the IL description shows that the user needs to know X to be able to do Y; there is no way they can know X; this is a problem! Others demand a deeper understanding of the underlying cognitive theory. That theory is based on the kind of means-end reasoning originally described by Newell and Simon[7]. Some design issues only become apparent if the analyst, writing an IL description, understands the consequences of the description, in terms of the implications for running models.

[7] See NEWELL, A. AND SIMON, H. (1972). *Human Problem Solving*, Englewood Cliffs, NJ: Prentice Hall.

3. Overview of the PUMA method

The Programmable User Modelling Analysis (PUMA) approach to user modelling considers the design of a computer system in terms of the tasks which the system is designed to support, and what the user is expected to do to achieve those tasks. The user's cognitive architecture is considered as a problem-solving mechanism of intentionally limited power — i.e. we don't want to assume the user can solve really hard problems; we want to say users are normal human beings, with limited problem-solving abilities. The notional target of the analysis is to produce a running simulation model of interactive behaviour, but the focus of the analysis is on describing user knowledge and device state, as a means of identifying potential inconsistencies between them.

The analysis consists of the following stages. As in most design and analysis activity, these stages do not follow each other in strict sequence, but provide a basic structure for the analysis activity.

1) Identifying candidate tasks:

The first stage is to identify candidate tasks for analysis by reference to the tasks the device is intended to support. These tasks are ones that have both domain and device relevance; that is, they are not tasks that relate solely to the device (e.g. "press a button"), but neither are they tasks that involve substantial knowledge about the context of use or organisational goals. Such tasks are typically quite small, and described at a detailed level. Examples might include undoing some typing in a multi-user editor[8] or pasting multiple copies of a piece of text within a word processing system. In this section, the tasks we consider are navigating around Web pages by using "hot" links and the history mechanism and a text-editing task. In the follow-on section we introduce some more substantial tasks. The selection of suitable tasks is based largely on craft skill, with the aim of choosing tasks that a user might find difficult for one reason or another.

2) Identifying conceptual operations:

The second stage is to identify *conceptual operations* to perform the candidate tasks. A conceptual operation corresponds to an action that is selected under certain conditions to achieve a particular purpose. For example, the task of deleting some text in an electronic document is achieved by marking the text and then either deleting it or cutting it. The "delete text" operation would be described in terms of "pressing the delete key to remove the text in the situation where the text is no longer needed", while the "cut text" operation would be described in terms of "selecting the *cut* command to remove the text in the situation where the text is to be used elsewhere". All the contextual information — such as what the preconditions of an operation are — is as important as the specification of the device action. The difficult, but important, thing to keep in mind is that these are *conceptual*; that is, they refer to the things the user is trying to achieve in the domain, rather than focusing solely on device actions.

3) Describing the user's knowledge in the Instruction Language:

The next stage is to describe the knowledge the user needs in an Instruction Language (IL), as described below. It also involves describing the device in similar terms.

One important focus of the analysis while writing the description is defining how users know things. Users might know things by coming to the interaction already knowing them, by looking at the screen, by tracking the effects of commands, or by making inferences from what is already known.

The process of writing this description may highlight sources of potential difficulty. Firstly, it may become apparent that it is not possible for users to know particular things that they would need to know to use the device effectively. Secondly, while experts may be reasonably expected to know particular information, novice users might not; laying out the knowledge needed in the Instruction Language helps the analyst to identify what needs to be known, and to consider how the information is best made available to novices (e.g. through training, or in a manual). One

---

[8] See YOUNG, R. M. AND ABOWD, G. (1994) Multi-perspective modelling of interface design issues: Undo in a collaborative editor. In G. COCKTON, S. W. DRAPER & G. R. S. WEIR (Eds) *People and Computers IX: Proceedings of HCI'94*. CUP: Cambridge, pp. 249-260.

general heuristic to follow when writing an IL description is that it should be kept as simple as possible.

As a constrained language, the IL limits what can be expressed formally. If important aspects of the design or user knowledge cannot be expressed precisely in the Instruction Language then they should be added as annotations. Such a description clearly cannot be converted faithfully into a running model (see stage 5), but can be used as a basis for reasoning and hand simulation.

If the description of the candidate tasks in terms of the IL helps to identify serious usability problems regarding what users need to know, then the analysis will typically stop here with a detailed account of the source of the difficulties, and possibly some proposals for remedying them.

If no such difficulties are found then the analyst can proceed to hand simulation.

4) Hand simulation of the model:

Hand simulation may be done for just one operation or for an extended task. It involves giving an account of how users acquire goals, become committed to operations, execute actions, and update their knowledge of the device state. The modelled problem solving is based on means-ends planning (identifying operations to address goals).

Hand simulation can also be done for just one modelled user or for users with different knowledge (for example, novice and expert). The changes in user knowledge over an extended interaction can be hand-simulated; this may identify points where the user's knowledge of the state of the device can get out of step with the actual device state; for example, the user may fail to track the effects of operations under certain circumstances or may track them inappropriately.

One aspect of modelling involves considering whether the effect of a command matches the user purpose. This is to ascertain whether the effect is predictable to the user, and whether there are circumstances in which there are potential mismatches between the user's intention and the device effect.

5) Running the model:

The IL description can be considered as a programming language for a Programmable User Model which, when compiled, yields a runnable cognitive model (see Appendix 1 for an example trace from a running model). The behaviour of this model can be compared with intended user behaviour; if there is a mismatch, then the analyst can refer to the trace of behaviour to identify the cause of the mismatch.

The aim of this enterprise is not to construct an artificial user (with all the real-world knowledge such a user would typically bring to bear on the tasks in hand), but to provide the designer or analyst with a means of identifying minimal requirements on the user's knowledge and capabilities.

In practice, constructing a running model generally serves as a notional, rather than an actual, target of the analysis, since the cost of constructing one for any sizeable problem outweighs the likely benefits.

Discussion

PUMA supports usability evaluation through the requirement to specify user knowledge clearly, to give an account of where that knowledge comes from and to compare user knowledge with device commands, and also through the provision of a way of reasoning about interactive behaviour.

However informal the analysis, the mode of working is guided by the requirements of a full analysis. In particular, the knowledge analysis is guided by the requirements of the Instruction Language, which in turn is designed as a programming language for the PUM architecture. The IL provides a language for description, and a tool to support critical thinking in design. Describing a design in terms of the IL also allows the analyst to develop the analysis further, to a full running model if appropriate, using the existing IL description as a basis; in this way, the analysis can be done incrementally.

Summary of Instruction Language

The IL is a declarative language for describing the knowledge users need to achieve any task with a device. The IL analyst's job is to describe what the user needs to know, and how any necessary

information is made available to the user, so that the user can both perform the task and know when the goal has been achieved.

The IL description consists of three parts:

- a set of declarations, listing
  - — the *conceptual objects* that the user is manipulating when working with the device, and
  - — *relationships* between those objects[9].
- a description of the user's knowledge, consisting of
  - — *conceptual operations*, which embody the knowledge the user needs about actions and effects,
  - — the user's *initial knowledge*, and
  - — the user's *task*, in terms of relations that should hold true in the goal state.
- a device description, listing
  - — the *device commands* that are available, in terms of the way the device state changes as a result of the user issuing each command,
  - — the initial *device state*, in terms of relations that hold true in that state, and
  - — what information is *displayed* to the user.

We need to expand on the various kinds of knowledge that must be included in conceptual operations. For *every* operation, we need to say:

- why the user would select this operation — i.e. what its *user purpose* is, and
- what the corresponding *action* is[10].

For most operations, there will be additional important contextual information, notably:

- any *arguments*, or parameters, to the operation — i.e. any variables in the context that it refers to (e.g. a selected piece of text or a particular page in a web browser);
- any *subgoaling preconditions* — that is, conditions that need to be true before the operation will achieve its intended effect, and which the user should plan to make true (e.g. pressing "delete" will only delete a particular item of text if that text is already marked);
- [a] any *filtering preconditions* — that is, conditions that need to be true before the operation will achieve its intended effect, but that the user either cannot or will not make true (e.g. if the user is building a red tower but has no red paint, then he can only pick up red bricks, so it would be a filtering condition that the brick must be red; note that if there is paint available then this condition might be a subgoaling precondition — i.e. the user might commit to selecting this brick, but would then adopt the goal of having it red first);
- any *predicted effects* — that is, effects of the operation that the user can predict reliably, without necessarily looking at the display (most important for effects that cannot be observed, such as copying text to a hidden buffer).

The use of these different kinds of knowledge will be illustrated through the examples in later chapters.

One of the important considerations for the analyst constructing an IL description is how users know all the things they need to know in order to achieve their goals with this device. There are three sources of information: facts that the user knew at the beginning of the interaction and that do not change (e.g. the user of an automated teller machine must know the relevant personal identification number), information that is available from the display, and information that must be predicted to be known. In many cases, an analysis that focuses just on this concern will highlight many of the usability problems with a device — there are things that the user clearly needs to know to work effectively with the device, but the information is not readily available from anywhere. In some instances, additional insights might be gained from hand-simulating the problem solving and external behaviour of the modelled user. Such hand-simulation can also be a good way to validate the IL description, as discussed below.

---

[9] Elsewhere, notably in most published papers, we make a distinction between two types of relationships, which we call "functions" (which are single-valued relations) and "predicates" (which are multi-valued). For the sake of simplicity, we avoid that distinction here.

[10] Elsewhere, we sometimes refer to this as a "device-command".

4. Example of using IL: Web navigation

To illustrate the use of the IL, we start with a small example, based on the use of a Web browser such as Netscape™ Navigator.

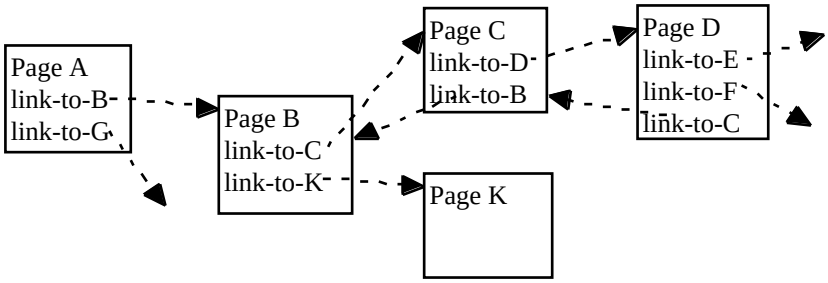


Figure 3: an example of a small section of Web, including links

To illustrate the use of a Web browser (particularly in case any readers are not familiar with Web navigation), imagine we have a small section of Web as illustrated in Figure 3. If viewing page A, the user can move to display page B (instead of A) by clicking on the link text “link-to-B”. Since there is no explicit link from B to A, the only way the user can return to viewing A is to press the “Back” button at this point. If she wanted to return again to B, she would have a choice: to press “Forward” (which “undoes” a “Back”), or to select the link to B again.

Consider a particular pattern of navigation: the user moves through pages A -> B -> C -> D -> C -> B -> K (Figure 3). What happens now if she presses “Back” twice?

> Consider your answer to this question; think about what knowledge about the device you are using to arrive at your answer. Then turn over the page and see what the “right” answer is.

As anyone who has studied Web navigation[11] will realise, the answer is "it depends". This example does not provide enough information to predict the outcome accurately. If the user has done all the navigation using links then the answer is the most obvious one : the user will move back to B then back to C. If, however, the user returned from D to C to B by pressing "Back" then the effect of pressing "Back" twice from K will be to move back to B and then to A. The reason for this is that the history list is maintained as a stack, and if a new item is added to the stack when the pointer is not at the top then all items above the current pointer position are removed (in this case pages C and D in the history list) and the new item (K) is added. The history list then contains A - B - K, so pressing "Back" twice moves the pointer from K through B to A (see Figure 4).

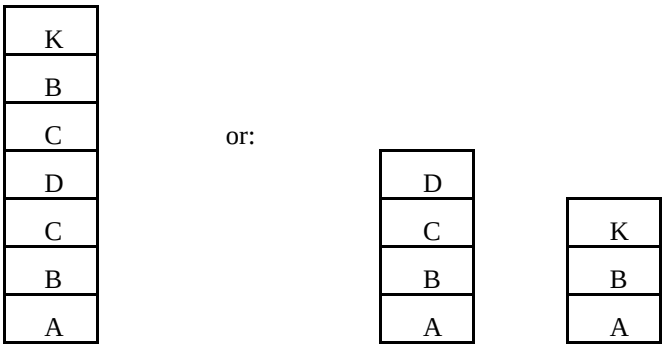


Figure 4: alternative possible history stacks for this sequence of pages

To be able to predict the effect, you have to have a sophisticated understanding of how the history list is maintained by the browser (as a "lossy" stack), of how "Back" works in relation to the history list, and also of the particular actions that the user performed to navigate from page D back to page B.

So far, we have only considered what knowledge you, as a "fly on the wall" have to have about this device to be able to make a prediction about the effect of the user's actions. We are more concerned, though, with what knowledge that user has, and with how they deploy their knowledge to perform tasks. So let's start laying that out.

The first step of doing an analysis is to define some candidate tasks. In this case, we will consider just one task scenario: the user has been navigating through several pages, using links and the "Back" and "Forward" buttons; while navigating, she noticed another link label that was relevant to her, and now she wishes to follow that link, but it was on a page a few pages back from her current position.

The user is working with entities of various kinds, including pages and links. We can list both object types and (for a particular scenario) relevant object instances. Let's assume that the pages are linked as shown in Figure 3, and that the user's task involves comparing information that is available on pages K and D. Let us say she is now viewing page D, and wishes to follow the link from B that points to K (except that she doesn't know that it points to K; all she knows is that the link label looks interesting).

```
OBJECTS
      page: A, B, C, D, E, F, … K, …
      link: AB, BC, CD, DE, DF, DC, CB, BK, AG, …
```

To describe the state of the device, or the user's knowledge, we need to be able to talk about how these objects relate to each other in general terms — i.e. what kinds of relationships between these object types are possible? The obvious relationships are that a link joins one page to

[11] See, for example, DIX, A. AND MANCINI, R. (1998), Specifying history and back tracking mechanisms. In Palanque, P. and Paterno, F. (Eds) *Formal methods in human-computer interaction.* pages 1--23. Springer Verlag., or

TAUSCHER, L. AND GREENBERG, S. (1997), How people revisit web pages: empirical findings and implication for the design of history systems, *International Journal of Human-Computer Studies* **47** pages 97--137.

another and that a particular page is displayed.

```
RELATIONS
      joins-page(link, page, page)
      is-displayed(page)
```

We may want to add further domain-oriented objects and relationships as we work through the analysis.

The initial device state can be described in terms of these relationships:

```
INITIAL DEVICE STATE
      joins-page(AB, A, B), joins-page(BC, B, C) … etc.
      is-displayed(D)
```

…and the available device commands can be described. We will consider the use of just "Jump" (to follow a link), "Back" and "Forward". It is not possible to describe the effects of "Back" and "Forward" accurately without also introducing the idea of an ordered (or indexed) history-list. We will represent the current device state by saying that particular pages have particular indexes, and that one of these is the index of the current page (i.e. the page that is displayed):

```
OBJECTS
      index: 1, 2, 3, …
RELATIONS
      in-history(index, page)
      current-index(index)
INITIAL DEVICE STATE
      in-history(1, A), in-history(2, B), in-history(3, C),
      in-history(4, D), current-index(4)
```

Then the available device commands have an effect on which page is currently displayed and also what the current index is:

```
DEVICE COMMANDS
      JUMP(LINK)
            if P0 is the initially displayed page and I0 is its
            index and joins-page(LINK, P0, P) then JUMP(LINK) has
            effect:
            is-displayed(P);
            in-history(I0+1, P);
            current-index(I0+1).
      BACK
            if P0 is the initially displayed page and I0 is its
            index and I0>1 and in-history(P, I0-1) then BACK has
            effect:
            is-displayed(P);
            current-index(I0-1).
            If I0=1 then BACK is not a valid device command.
      FORWARD
            if P0 is the initially displayed page and I0 is its
            index and in-history(I0+1, P) [i.e. there is a page
            forward in the history list] then FORWARD has effect:
            is-displayed(P);
            current-index(I0+1).
            If there are no pages forward in the history list
            then FORWARD is not a valid device command.
```

To readers who are unfamiliar with such history mechanisms, this device description may appear to be rather complicated. From a device perspective, however, it is a fairly simple abstract description; the challenge now is to assess how such a device is likely to be used in practice.

We have quite a lot of scope in defining the user's knowledge. For the purposes of this initial example, we will make the scenario very concrete: the user has navigated from page A, following links that seem to point to pages that are likely to be relevant to the topic of interest, and is now at D. On the way, there appeared to be another link from page B that might also be relevant, so now the user wants to go there. How do we describe the user's side of this scenario in the Instruction Language? The user's goal is to be in a state where she is viewing a page that

contains the relevant information. There are various ways we might choose to represent this. We will opt for one that is moderately device-oriented, rather than one that would force us to add too much knowledge about the relationship between the device task (get to a particular page, or pages) and the domain task (find out particular information). We say that the desired state is one in which the page displayed is one that is linked to B by a link that appears to be interesting:

```
DESIRED STATE
      is-displayed(page) such that joins-page(link, B, page)
```

We are also going to need relationships about pages and links the user remembers seeing:

```
RELATIONS
      have-visited(page), have-seen-interesting(link)
USER KNOWS INITIALLY
      is-displayed(D)
      have-visited(A), have-visited(B)
      have-visited(C), have-visited(D)
      have-seen-interesting(BK)
```

Now we need to define just one more kind of information: the user's knowledge of operations. This is a way of relating what a user might reasonably be expected to know about the domain and device with the relevant device commands.

For the purpose of this scenario, we only need to consider two operations, which we will call "follow-link" and "retrace-steps-to-find-link".

For the first of these, we wish to express the idea that the purpose of the operation is to display some page (as yet unknown), about which the only thing the user knows is that the link label looks relevant to the topic of interest. This can only be done if the relevant page (containing the link) is displayed, and then the appropriate action is to follow the link (or "jump"). The second operation expresses typical user knowledge about the use of "Back" to get to a desired link — that the purpose is to see (again) some page that has already been seen, that had an interesting link on it. We could describe a very similar operation that describes the knowledge the user would need to revisit a page in order to review information that was displayed on that page. We leave it as an exercise to the reader to define such an operation; we simply note here that the reason for making a distinction is that the user applies different knowledge in the two cases, and that this distinction might be important in some circumstances.

```
OPERATIONS
follow-link(link: L, page: Pany, page: Punknown)
      user-purpose:           is-displayed(Punknown)
      filtering-precond:      joins-page(L, Pany, Punknown)
      subgoaling-precond:     is-displayed(Pany)
      action:                 jump(L)

retrace-steps-to-find-link(link: L, page: Pany)
      user-purpose:           is-displayed(Pany)
      filtering-precond:      have-seen(L)
                              joins-page(L, Pany, Punknown)
      action:                 back
```

For every operation, it is essential to define its user-purpose (what the operation is *for*) and the corresponding action. In this case, we have also defined one *subgoaling precondition* — that is, a condition that the user might plan to make true in order to achieve a desired effect — and some *filtering preconditions* — that is, conditions that influence the user's choice of behaviour, but that the user cannot or would not try to make true explicitly. In this case, we are assuming that the user cannot change the structure of the Web, or whether or not a particular link is interesting, but they can change which page is displayed.

We are now in a position to trace out by hand how the specified model will behave. The model has two core principles that drive its behaviour:

> *Selection by purpose*: If the model has a goal G, and if there is an operation O with a user-purpose that matches G and whose filtering preconditions are satisfied, then the model will choose O — or one of those operations, if there are several.
>
> *Precondition subgoaling*: If the model has chosen an operation O, and if it has

subgoaling precondition(s) P that are not satisfied, then the model stays committed to executing O but first sets up the goal(s) of achieving P.

Such principles cannot capture the richness and diversity of natural human behaviour, but provide a starting point for an analysis of simple interactive behaviour. The modelled behaviour also takes account of changes in the device state. When an operation with satisfied preconditions is selected, the user issues the corresponding device command and the user's knowledge of the device state changes in response to predicted information and perceived changes to the device state (Figure 5).

In this example, the user does not have to predict the effects of any operations because all the relevant information is available from the display; all the user has to do is remember that they have seen something relevant before.

The following table illustrates how the modelled user acquires goals and commitments, and selects actions to achieve goals. Once the user becomes committed to an operation whose preconditions are satisfied, the user can start acting, and through visible effects the user's knowledge of the state of the device is updated until the goal of the task is achieved.

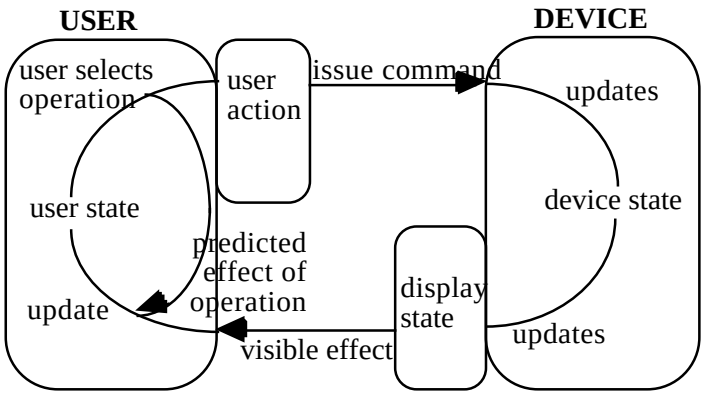


Figure 5: separate user and device models

Device state	User's knowledge of state	User's goals	User's commitments	Notes
joins-page(AB, A, B), joins-page(BC, B, C) … etc. is-displayed(D) in-history(1, A), in-history(2, B), in-history(3, C), in-history(4, D), current-index(4)	is-displayed(D), have-visited(A), have-visited(B), have-visited(C), have-visited(D), have-seen(BK)	is-displayed (Punknown) such that joins-page(BK, B, Punknown)		The user's goal matches the user-purpose and filters of follow-link(BK, B, Punknown), so user becomes committed to this operation
"	"	"	follow-link(BK, B, Punknown)	This has a subgoaling precondition, so the user adopts the goal
"	"	as above + is-displayed(B) such that joins-page(BK, B, Punknown)	"	The new goal matches the user purpose and filters of retrace-steps-to-find-link(BK, B), so the user becomes committed to this
"	"	"	add retrace-steps-to-find-link(BK, B)	This operation has no subgoaling preconditions, so the user performs the action "Back".
delete is-displayed(D); add is-displayed(C); delete current-index(4); add current-index(3)	delete is-displayed(D); add is-displayed(C)	"	delete retrace-steps-to-find-link(BK, B)	The action is performed so the user ceases to be committed to the operation. The user can see the effect of their action on the display, so updates knowledge accordingly.

joins-page(AB, A, B) … etc. is-displayed(C) in-history(1, A), in-history(2, B), in-history(3, C), in-history(4, D), current-index(3)	is-displayed(C) …etc.	"	add retrace-steps-to-find-link(BK, B)	Goal and knowledge still match same operation, so commit again and perform the action "Back"

delete is-displayed(C);
add is-displayed(B);
delete current-index(3);
add current-index(2)
	delete
	is-displayed(C);
	add is-displayed(B)
		delete
		is-displayed(B) such that
		joins-page(BK, B, Punknown)
			delete
			retrace-steps-to-find-link(BK,B)
				subgoal now achieved, so it's removed from the goal stack

joins-page(AB, A, B) … etc. is-displayed(B), … current-index(2)	is-displayed(B) …etc.	is-displayed(page) such that joins-page(BK, B, page)	follow-link(BK, B, Punknown)	precondition of follow-link operation now satisfied, so perform corresponding action "Jump(BK)"
joins-page(AB, A, B) … etc. is-displayed(K) in-history(1, A), in-history(2, B), in-history(3, K), current-index(3)	is-displayed(K), have-visited(A), have-visited(B), have-visited(C), have-visited(D), have-visited(K), have-seen(BK)	goal achieved	commitment dropped	

In this example, we have introduced the ideas that:

- the modelled user selects actions to achieve goals, on the basis of what is currently known; and
- the user's knowledge of the device state is updated through looking at the device.

We leave it as an exercise for the reader to define a second scenario, starting from this situation, to hand-simulate the consequences of the user now deciding that they want to follow link DF. You will start by adding the knowledge have-seen(DF), and define a new goal. We hope it is obvious even before you start that with the given knowledge, the user will fail to achieve their goal. You might consider how the design of the interface, or of the history mechanism, might be modified to alleviate this particular problem. You should be warned, though, that any alternative implementation of a history mechanism probably has other usability difficulties of one sort or another…

We have not covered all features of the IL in this example. In particular, this example doesn't force the user to keep track (in her head) of the effects of any actions. the next example includes a case of this — that to be able to use the contents of a hidden buffer, the user has to know what is in there, and that the user can only know that by knowing what they put (cut or copied) in there most recently.

In the Web example you might try to model a sophisticated user who knows about the history mechanism, and can therefore keep track of what is in the history list without pulling down the relevant menu; however, for all bar the shortest interaction sequences the memory load required is unreasonably large.

Another exercise you might try is to specify the user's knowledge about "Forward". It turns out to be quite difficult to specify a user-purpose and other conditions such that the user would select forward. This is consistent with empirical results, which show that users make very little use of the "Forward" button.

Our answer to the second scenario is as follows:

The user has seen a link label that is interesting and wishes to see the page that is reached by following it:

```
DESIRED STATE
      is-displayed(page) such that joins-page(DF, D, page)
```

The user has seen DF:

```
USER KNOWS INITIALLY
      have-seen(DF)
```

Then, just as in the table above, the user knows that the way to achieve the goal is to follow the link from the page that has been seen before, which means that the page has to be displayed. So the user becomes committed to displaying the page. The user knows (or, more accurately, believes) that the way to achieve this is by retracing steps to find the link. The user therefore presses “back”, but this time the operation does not achieve the goal (instead, it takes the user to B and then A). To allow this user to succeed, we would have to add knowledge about “back” being appropriate only if the target page is in the history list, or if it is more easily reached from a page in the history list than from the current page. We would also have to add knowledge about a new operation — to follow links previously followed to retrace steps from a section of history that has been lost from the history stack. This is clearly much more sophisticated knowledge than that currently expressed, reflecting the difficulty of using the history mechanism in many cases.

5. Example: cutting and pasting in a text editor

Our second example is a familiar interactive device: a text editor. We are using this example to pull out more details about constructing an appropriate IL representation. Ultimately, there is no uniquely "correct" IL description of any device or scenario, but some descriptions are easier to construct and reason with than others. It is this issue — of choosing a "good" representation — that we focus on in this example. We are also trying to convey the iterative nature of IL construction, so several times we propose a description that we later withdraw, to illustrate some of the thinking that goes into IL construction.

We consider an editor that has commands to:

- copy text to a hidden buffer,
- cut text from the document (into the hidden buffer), and
- paste text from the hidden buffer.

Text to be copied or cut first has to be marked, and the cursor has to be located on text before that text can be marked.

Pasting text causes a copy of the text in the hidden buffer to be inserted at the current cursor position (Figure 6).

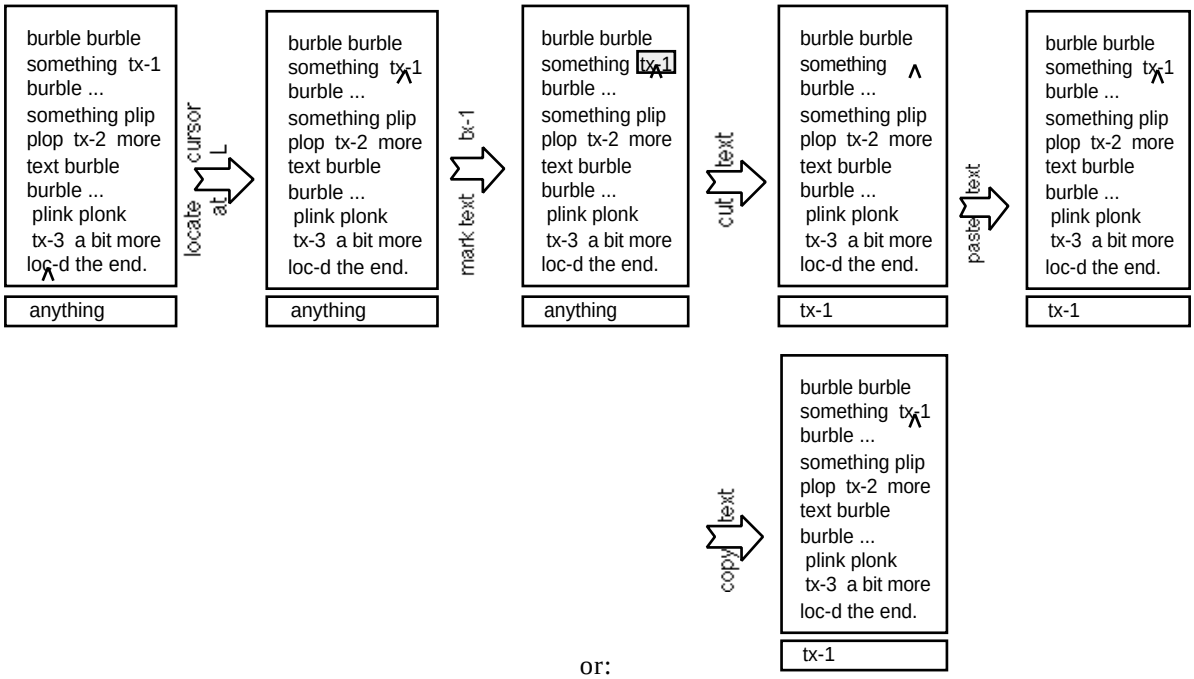
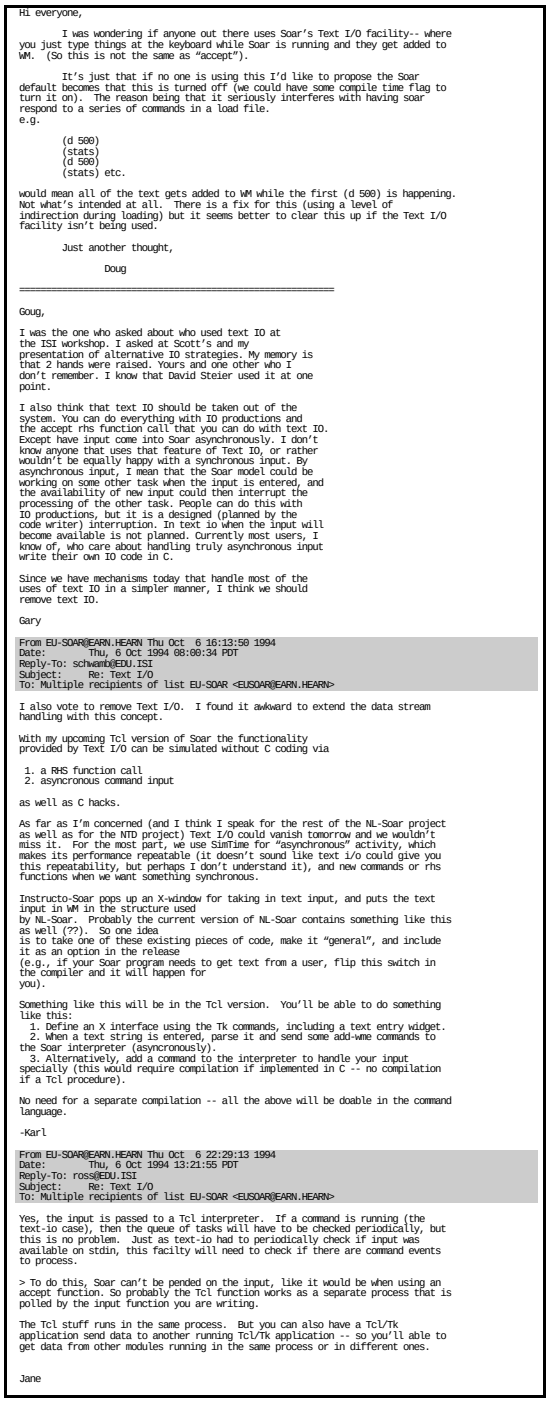


Figure 6: effects of device commands on an example document

As in every analysis, we start by identifying candidate tasks. In this case, we propose just one task scenario. We imagine that the user has saved several mail messages (with their headers) to a file. She wants to replace all the headers with dashed lines. She has already replaced the first one; now she intends to use the cut and paste facilities to change the others (Figure 7).



Hi everyone,

I was wondering if anyone out there uses Soar's Text I/O facility-- where you just type things at the keyboard while Soar is running and they get added to WM. (So this is not the same as "accept").

It's just that if no one is using this I'd like to propose the Soar default becomes that this is turned off (we could have some compile time flag to turn it on). The reason being that it seriously interferes with having soar respond to a series of commands in a load file.
e.g.

(d 500)
(stats)
(d 500)
(stats) etc.

would mean all of the text gets added to WM while the first (d 500) is happening. Not what's intended at all. There is a fix for this (using a level of indirection during loading) but it seems better to clear this up if the Text I/O facility isn't being used.

Just another thought,

Doug

==========

Goug,

I was the one who asked about who used text IO at the ISI workshop. I asked at Scott's and my presentation of alternative IO strategies. My memory is that 2 hands were raised. Yours and one other who I don't remember. I know that David Steier used it at one point.

I also think that text IO should be taken out of the system. You can do everything with IO productions and the accept rhs function call that you can do with text IO. Except have input come into Soar asynchronously. I don't know anyone that uses that feature of Text IO, or rather wouldn't be equally happy with a synchronous input. By asynchronous input, I mean that the Soar model could be working on some other task when the input is entered, and the availability of new input could then interrupt the processing of the other task. People can do this with IO productions, but it is a designed (planned by the code writer) interruption. In text io when the input will become available is not planned. Currently most users, I know of, who care about handling truly asynchronous input write their own IO code in C.

Since we have mechanisms today that handle most of the uses of text IO in a simpler manner, I think we should remove text IO.

Gary

From EU-SOAR@EARN.HEARN Thu Oct 6 16:13:50 1994
Date: Thu, 6 Oct 1994 08:00:34 PDT
Reply-To: schwamb@EDU.ISI
Subject: Re: Text I/O
To: Multiple recipients of list EU-SOAR <EUSOAR@EARN.HEARN>

I also vote to remove Text I/O. I found it awkward to extend the data stream handling with this concept.

With my upcoming Tcl version of Soar the functionality provided by Text I/O can be simulated without C coding via

1. a RHS function call
2. asyncronous command input

as well as C hacks.

As far as I'm concerned (and I think I speak for the rest of the NL-Soar project as well as for the NTD project) Text I/O could vanish tomorrow and we wouldn't miss it. For the most part, we use SimTime for "asynchronous" activity, which makes its performance repeatable (it doesn't sound like text i/o could give you this repeatability, but perhaps I don't understand it), and new commands or rhs functions when we want something synchronous.

Instructo-Soar pops up an X-window for taking in text input, and puts the text input in WM in the structure used by NL-Soar. Probably the current version of NL-Soar contains something like this as well (??). So one idea is to take one of these existing pieces of code, make it "general", and include it as an option in the release (e.g., if your Soar program needs to get text from a user, flip this switch in the compiler and it will happen for you).

Something like this will be in the Tcl version. You'll be able to do something like this:
1. Define an X interface using the Tk commands, including a text entry widget.
2. when a text string is entered, parse it and send some add-wme commands to the Soar interpreter (asyncronously).
3. Alternatively, add a command to the interpreter to handle your input specially (this would require compilation if implemented in C -- no compilation if a Tcl procedure).

No need for a separate compilation -- all the above will be doable in the command language.

-Karl

From EU-SOAR@EARN.HEARN Thu Oct 6 22:29:13 1994
Date: Thu, 6 Oct 1994 13:21:55 PDT
Reply-To: ross@EDU.ISI
Subject: Re: Text I/O
To: Multiple recipients of list EU-SOAR <EUSOAR@EARN.HEARN>

Yes, the input is passed to a Tcl interpreter. If a command is running (the text-io case), then the queue of tasks will have to be checked periodically, but this is no problem. Just as text-io had to periodically check if input was available on stdin, this facilty will need to check if there are command events to process.

> To do this, Soar can't be pended on the input, like it would be when using an accept function. So probably the Tcl function works as a separate process that is polled by the input function you are writing.

The Tcl stuff runs in the same process. But you can also have a Tcl/Tk application send data to another running Tcl/Tk application -- so you'll able to get data from other modules running in the same process or in different ones.

Jane

<= copy this...

<= to replace this...

<= and this.

Figure 7: editing email messages scenario

Given what you have been told about the text editor, spend a few minutes sketching out ideas about how you would construct an IL description.

- what are the conceptual object types?
- are there individual objects you want to name?
- what relations do you need to define?
- what are the conceptual operations?

— do they have subgoaling preconditions or filtering preconditions?

— what about purpose? (what does the user do this "for"?)

A summary of the Instruction Language can be found in Appendix 3, at the back of this document.

Choosing a representation

We want to use a representation that's as *simple* as possible. For example, we are ignoring the fact that the document probably can't all be shown on the display at one time.

We use what we are told about the way the device works and about the task to help to construct an appropriate representation. For, example, we are told that:

- the device has commands to copy text to a hidden buffer, to cut text from the document (into the hidden buffer) and to paste text from the hidden buffer.
- text can be marked.
- the cursor has a position.

...so obvious things to represent include some way of describing text and places in the document. Also, there are a cursor and a hidden buffer. This leads us to tentatively specify some object types:

```
OBJECTS
text:
location:
cursor:
buffer:
```

Here, we have started from a general description of the device and the commands available, and generated some object types.

We need to go on and decide in more detail what these terms mean. For example, we could describe the text in terms of letters, words, lines or "chunks". We need to refer back to the task to select the most appropriate representation. In this case, since cutting and pasting work on arbitrary units of text, "chunks" are the best.

Then we need to consider what "chunks" need to be explicitly specified. We obviously need to be able to talk about the bits to be copied and the bits to be replaced. But we don't need to consider what's in them, so let us just give them arbitrary labels:

```
text: tx-1, tx-2, tx-3.
```

We might want to also consider the text of the rest of the document. Maybe what we want to say is that the document consists of these "chunks" of text, and then define what order they come in to start with, and what order we want them to finish up in. It will depend to some extent on how we represent locations. But maintaining ordered lists ("This bit of text immediately follows that bit") is often difficult, so if possible we will do without.

We need to reason in a similar was about how to represent locations in the document. Locations could be absolute screen locations (but how does that relate to the contents of the document?), fixed locations within the document (but what happens when you cut or paste text?), "between" two chunks of text (so when you add or delete chunks of text, you also add or delete locations in the document), or just arbitrary labels for significant (task-related) places in the document (but what happens if you paste multiple chunks of text at the same significant location?).

Suppose locations are "between" text chunks. Then we need to name all chunks of text, as shown in figure 8.

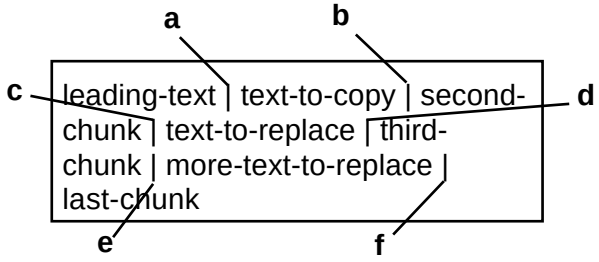


Figure 8: locations between chunks of text

If we were inserting additional chunks of text between existing chunks, we would need a representation such as this. However, the task we are considering today is simpler: to replace chunks of text with different chunks. For this task, we find that we can get away with using arbitrary labels to refer to "the place where this chunk of text is", labelled in Figure 9 as locations a, b, c, d.

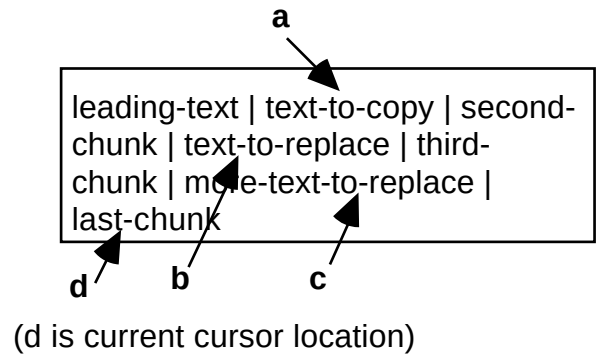


Figure 9: locations that are significant

Note that with this representation, we could have 0, 1 or more chunks of text at a location. This representation happens to be appropriate for this task, but might be less appropriate for many others.

To cut a long story short: we consider what we are interested in modelling, and choose a simple representation for everything else.

Given these simplifications, we can now describe the task schematically:

The user has a document that needs a couple of minor modifications. Two pieces of text — at locations B and C — need to be replaced by copies of a third piece, which is currently in the document at location A. The current situation is shown in Figure 10; we will assume the cursor is at D.

We now need to describe the task scenario in terms of initial and desired states. The task can be described in terms of adding chunks of text at locations and removing chunks of text from locations, so we obviously need a relation that says "chunks of text are at locations":

```
RELATIONS
text-at(text, location)
```

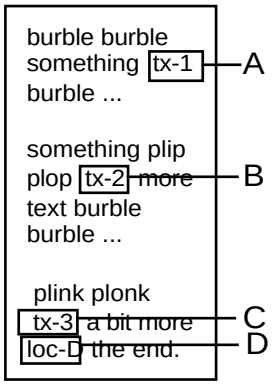


Figure 10: schematic representation

Then we can define the "state of the document" at the beginning:

```
INITIAL DEVICE STATE
text-at(tx-1,a)
text-at(tx-2,b)
text-at(tx-3,c)
```

... and the state of the document at the end in terms of things that have changed from the initial state (note that this includes a statement of things that should *not* be true in the desired state as well as things that *should* be true):

```
DESIRED STATE
text-at(tx-1,b)
not text-at(tx-2,b)
text-at(tx-1,c)
not text-at(tx-3,c)
```

We also need to define what the user knows about operations. One way to begin is to consider the set of device commands. Referring back to the original description, we see that these are:

```
action:          copy-to-buffer
action:          cut-text
action:          paste-text
action:          mark-text (TX)
action:          locate-at (L)
```

In this list, `L` is a location (the place where the cursor is moved to), and `TX` is a chunk of text (the chunk that is to be marked). These are *arguments* on the device command.

Each device command is there for a reason! We start by assuming that we need a conceptual operation to correspond to each, and enter all the things the user has to know about this operation.

Let us consider paste-text....*What's the point of pasting text?* — it is to get the desired text at the intended place in the document, so we can state that as the user-purpose:

```
user-purpose:     text-at(TX,L)
```

What has to be true before issuing a paste-text command will have this effect? — the correct text must be in the hidden buffer, and the cursor must be at the right place for pasting.

There are alternative ways we might say that the correct text must be in the buffer. For instance, we could use a relation to say:

```
contains(B, TX)
```

— where B is of type *buffer*. However, since there is only one buffer, there is little point in declaring it as an object. Instead, we can say:

```
buffer-contains(TX)
```

By similar reasoning (there is only one cursor; it has exactly one location), we can define a relation for the cursor position:

```
cursor-at(L)
```

When it comes to hand-simulation, there is an important difference between these two relations and the first (text-at): text-at can take many values simultaneously, whereas buffer-contains and cursor-at can only take one value (and must take precisely one value) at a time. For now, we do not worry unduly about this difference, but it can be important in some situations[12].

Earlier on, we defined some object types and a relation. We have now realised that:

- we don't need object types cursor and buffer;
- we do need two more relations to define the buffer contents and the cursor location.

We can now summarise the IL definition we have constructed so far:

```
OBJECTS
text: tx-1, tx-2, tx-3.
location: a, b, c, d.

RELATIONS
cursor-at(location).
buffer-contains(text).
text-at(text, location).

OPERATIONS
operation paste-text-at-loc (text:TX, location:L)
      user-purpose:           text-at(TX,L)
      subgoaling-precond:     buffer-contains(TX)
                              cursor-at(L)
      action:                 paste-text
```

[12] As noted earlier, we are presenting all RELATIONS in a uniform way in this tutorial. Elsewhere, we have distinguished between PREDICATES (which are many-valued relations) and FUNCTIONS (which are single-valued).

We are almost ready to complete definitions of more operations. But first, one more relation is needed so that we can talk about chunks of text being marked.

With our current simplified representation of locations, such that there might be more than one chunk of text at a location, we need to specify both the chunk of text and the location of that text. For this, we will use the relation:

```
is-marked(text, location)
```

Given the IL description that we have constructed so far, complete the following. (Note that some of the entries may be irrelevant.)

```
OBJECTS
      text: tx-1, tx-2, tx-3.
      location: a, b, c, d.
RELATIONS
      cursor-at(location)
      buffer-contains(text)
      text-at(text, location)
      is-marked(text, location)
OPERATIONS
operation locate-cursor (location:L)
      user-purpose:           ?
      subgoaling-precond:     ?
      action:                 locate-at (L)
operation mark-text-at-loc (text:TX, location:L)
      user-purpose:           ?
      subgoaling-precond:     ?
      filtering-precond:      ?
      action:                 mark-text (TX)
operation cut-text-from-loc (text:TX, location:L)
      user-purpose:           ?
      subgoaling-precond:     ?
      action:                 cut-text
operation copy-text-to-buffer (text:TX, location:L)
      user-purpose:           ?
      subgoaling-precond:     ?
      action:                 copy-to-buffer
operation paste-text-at-loc (text:TX, location:L)
      user-purpose:           text-at(TX,L)
      subgoaling-precond:     buffer-contains(TX)
                              cursor-at(L)
      action:                 paste-text
```

When you have completed your answer and are happy with it, please turn over the page to compare your answer with ours.

Our answer to this is constructed as follows:

Firstly, locate-cursor has no preconditions (the user can locate the cursor at any position desired on the page, since we are not considering multi-page documents):

```
operation locate-cursor (location:L)
      user-purpose:           cursor-at(L)
      action:                 locate-at (L)
```

Text can only be marked if the cursor is in the right place (so this is a subgoaling precondition); however, the user would not choose to put a chunk of text at a particular location just so that it can be marked (so this is a filtering precondition, or selection rule):

```
operation mark-text-at-loc (text:TX, location:L)
      user-purpose:           is-marked(TX,L)
      subgoaling-precond:     cursor-at(L)
      filtering-precond:      text-at(TX,L)
      action:                 mark-text (TX)
```

Text must be marked before it can be cut, and the purpose of cutting it is to remove it from the current location:

```
operation cut-text-from-loc (text:TX, location:L)
      user-purpose:           not text-at(TX,L)
      subgoaling-precond:     is-marked(TX,L)
      action:                 cut-text
```

Similarly, when copying text to the buffer it has to be marked, and the purpose of copying is to get the text into the buffer. However, in this case, we have to say something more: that the user *predicts* the effect of the action. Since the buffer contents are not displayed on the screen, the only way the user can know what is in there is by predicting the effects of actions:

```
operation copy-text-to-buffer (text:TX, location:L)
      user-purpose:           buffer-contains(TX)
      subgoaling-precond:     is-marked(TX,L)
      predicted-effect:       buffer-contains(TX)
      action:                 copy-to-buffer
```

When we do a hand-simulation of the interaction between user and device, we have to model how the device state changes, and also what the user knows about the device state.

When discussing web navigation, the user knew everything else she needed to because it was visible and she could see it. This is not always the case.

Now we make this more explicit:

> For everything that the user needs to know, there must be a way for him to know it.
>
> This can be:
>
> a) by interpreting something from the display,
>
> b) by tracking the effect of an operation, or
>
> c) by knowing it at the beginning of the interaction.

As outlined earlier, we can use this IL description for reasoning about likely interactive behaviours: if the user has particular knowledge, and applies particular problem-solving mechanisms to decide what to do next, and updates their knowledge of the system state in particular ways, then we can infer what interactive behaviours are possible on that basis.

The problem solving proceeds as follows: firstly, the (modelled) user identifies candidate operations to reduce the difference between the current and desired states. The user then selects one operation to perform. If it can be done now, the user does it; otherwise the user tries to satisfy the subgoaling preconditions of the operation. The user maintains knowledge of the state of the device partly by predicting the (known) effect of some operations and partly by getting information from the screen.

One of the designer's tasks is to consider how the user is expected to interpret information from the display. If the mapping from displayed information to user knowledge of the state of the

device is not a simple 1-1 mapping, the designer should make the mapping explicit. In simple cases, we can make a 1-1 correspondence (assuming the user can interpret the information as displayed in terms of what it means about the underlying system state). This is what we do for the text editor.

Then we define how the user knows things:

```
RELATIONS
cursor-at(location) -- detected by looking
buffer-contains(text) -- user must predict and remember
text-at(text, location) -- detected by looking
is-marked(text, location) -- detected by looking

OPERATIONS
operation copy-text-to-buffer (text:TX, location:L)
      user-purpose:           buffer-contains(TX)
      subgoaling-precond:     is-marked(TX,L)
      predicted-effect:       buffer-contains(TX)
      action:                 copy-to-buffer
                        ** user must remember buffer-contains(TX)
```

As well as describing the user's knowledge, we also have to describe the device model (to account for the device changes that result from user actions). The device model does not have to be specified as formally as the user model. Descriptively:

```
DEVICE COMMANDS
locate-at (L)
   The cursor location is updated to be L. Any marked text in the
   document ceases to be marked.
mark-text (TX)
   Text TX at the current location is marked.
cut-text
   The currently marked text is deleted from the document and put
   into the buffer.
copy-to-buffer
   The currently marked text is copied into the buffer.
paste-text
   The contents of the buffer appear in the document at the
   current cursor location.
```

We also have to state what the initial device state is, and what is displayed:

```
INITIAL DEVICE STATE
      text-at(tx-1,a)
      text-at(tx-2,b)
      text-at(tx-3,c)
      cursor-at(d)
      buffer-contains(text-unknown)

DISPLAYED
      Everything except buffer contents.
```

There is a further very important point about this description, to do with when the user tracks information. The observant among you might have noticed that in our definition of cut-text-from-loc we did not specify that the change to buffer contents was predicted:

```
operation cut-text-from-loc (text:TX, location:L)
      user-purpose:           not text-at(TX,L)
      subgoaling-precond:     is-marked(TX,L)
      action:                 cut-text
```

This is important! Users often make errors precisely because they fail to track side-effects of operations.

A heuristic: main effects of operations (the "user purpose") are predicted if they are predictable to the user — particularly if they are not visible — but side-effects are not predicted.

Note that an effect that is neither visible nor predictable is not usable by the user.

Hand simulation

The IL description is now complete, and we can reason about likely behaviours. In fact, in this case, we can look at the results of computer simulation (a running model), as shown in Appendix 1, as well as doing hand simulation (which should yield the same results).

Initially, there are four differences between the initial and the desired states.

List the differences between initial and desired states.

All of these differences are equally important, so there is no very obvious way to choose between them, and the modelled user could select any difference to remove first. Because of this, there are various behaviours this model could generate. Let us focus on one of them; a full trace of this behaviour is included in Appendix 1 (generated by producing a running model from this IL description). In this case, the first thing the modelled user commits to is getting a copy of tx-1 at location b, for which the appropriate operation is paste-text(tx-1, b). This operation has two subgoaling preconditions that are not yet satisfied (to be at the right place in the document, and to have the right text in the buffer), so the modelled user has to select one of these to address first. In practice, since moving the cursor around is "easier" than getting the correct text in the buffer, the model should automatically select the goal of getting the correct text into the buffer first; in fact this is manually selected by the analyst in this version of the program. Further subgoaling takes place until the model can actually do something (move the cursor to the correct place for copying text to the buffer), and from then on, the model proceeds to easily address outstanding goals until the text has been pasted at location b. There is now a choice of three outstanding goals to achieve. If the analyst selects removing tx-2 from b as the next goal then the behaviour proceeds as in figure 11. As shown in the trace (point marked by side-bars), this results in a mis-match between the actual device state and the user's knowledge of that state, as the user fails to predict the side-effect of cutting text [that it overwrites whatever is in the buffer].

The running model does not implement error recovery, so at the point where Figure 11 shows the user realising the mistake, the running model just doggedly keeps trying to re-paste the wrong text at location b.

This modelled behaviour is just one of the space of possible behaviours; in principle, all of them can be analysed, either by re-running the model, or by hand-simulation.

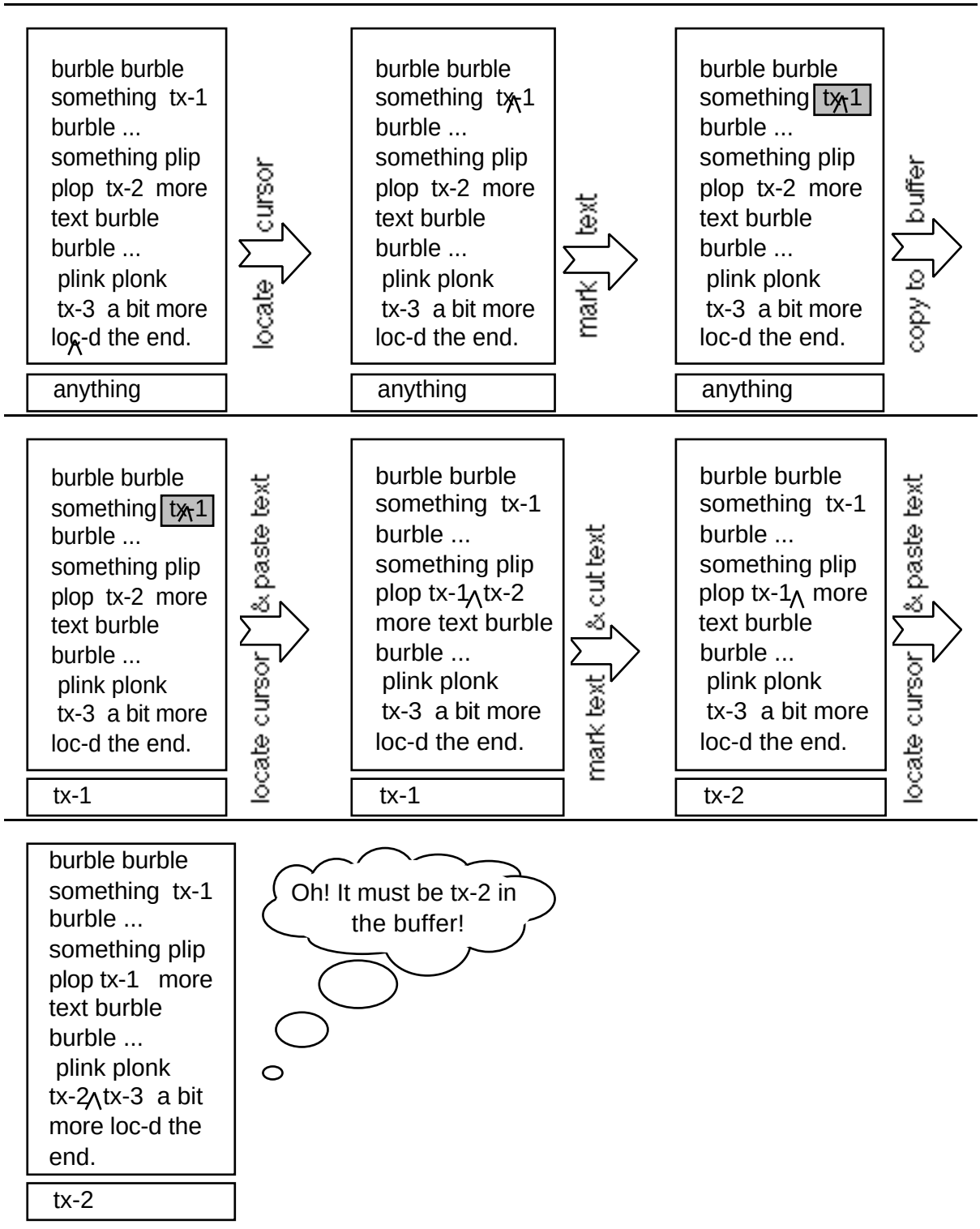


Figure 11: surface behaviour of model

Variant on the task

Let us now consider a variant on this task. Suppose that, starting from the same initial state, the user now wants to replace the text at B with the text from A (moving it from A).

We can ignore location C this time, so we now have initial and desired states:

```
INITIALLY
text-at(tx-1,a)
text-at(tx-2,b)

DESIRED-STATE
not text-at(tx-1,a)          (tx-1 no longer at a)
text-at(tx-1,b)              (tx-1 instead of...
not text-at(tx-2,b)          ......tx-2 at b)
```

This time, the user will be doing something slightly different: she will be moving tx-1 from A into the hidden buffer.

> Describe the operation that is needed for this. Note that in this case, the user purpose is to achieve two effects simultaneously, and the action is one that is already the action corresponding to another conceptual operation.
>
> Do a hand-simulation to establish what the space of possible behaviours is in this case, and identify the worst possible behaviour.
>
> Do not turn over the page until you are happy with your answer!

The operation to move text to the buffer can be described as follows:

```
operation move-text-to-buffer (text:TX, location:L)
      user-purpose:           buffer-contains(TX) &
                              not text-at(TX,L)
      subgoaling-precond:     is-marked(TX,L)
      predicted-effect:       buffer-contains(TX)
      action:                 cut-text
```

There are various behaviours this model could generate. The worst of them is shown in figure 12.

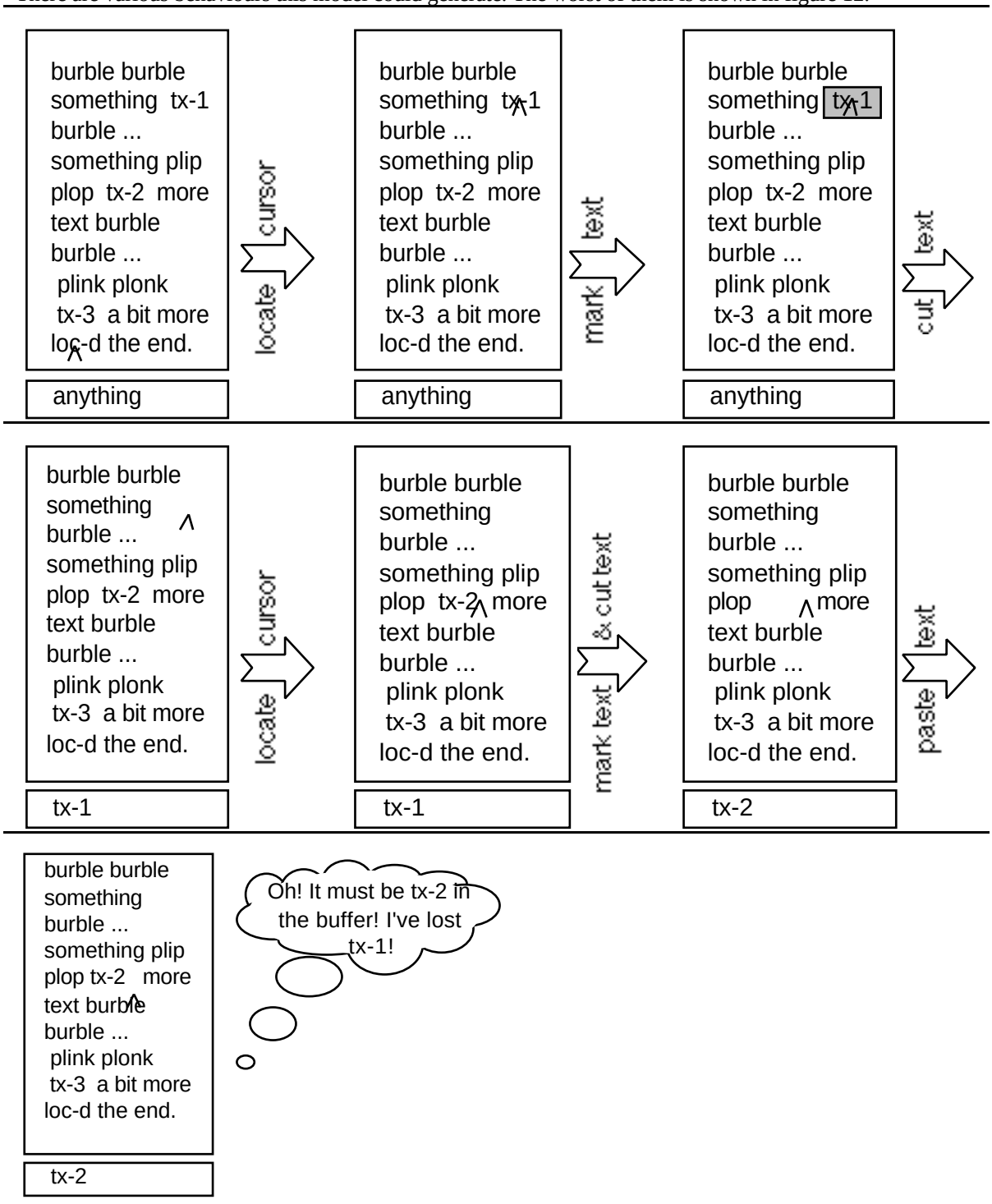


Figure 12: behaviour of the second running model

We can summarise the important issues in relation to the design of the text editor as follows:

- For the first task, plausible sequences of actions include getting tx-2 pasted at C instead of tx-1. This error is recoverable. (Delete that text again, go back to B, copy tx-1 again, return to C and paste-text.)
- For the second task, plausible sequences of actions include over-writing tx-1 in the hidden buffer, then getting tx-2 pasted at B instead of tx-1. This error, while not inevitable, is plausible and not recoverable.

More general observations relating to this analysis include the following:

1) User should not have difficulty with locating the cursor, or marking or copying text.
2) Users may fail to predict the side-effect of *cut*, and so *paste* the wrong piece of text.
3) More seriously, in failing to predict the side-effect of a *cut*, users may accidentally over-write the contents of the hidden buffer before re-pasting text back into the document.
4) We surmise that designers implemented *cut* in this way (e.g. on Sun workstations under OpenWindows) so that the user could recover material deleted from the document. This implementation makes it easy to, for example, swap two pieces of text.
5) These particular problems might be avoided by having an implementation of *cut-text* that simply removes the text completely (retrievable with an *undo* operation), distinct (in terms of device command) from *cut-to-buffer*. This is the approach taken by the designers of the Mac interface, for which *delete* is distinct from *cut*.

To summarise, in this example, we have shown in detail how the IL description is constructed, how it is used in hand-simulation, and how it can be used for reasoning about potential usability difficulties (as well as things users are likely to find easy!) We have shown how the device state and the user's knowledge of that state can get out of synch, and how different conceptual operations can correspond to the same device action. These are all important points to bear in mind in future analyses.

6. Closing discussion: section 1

In this section, we have introduced the PUMA approach to cognitive modelling and usability evaluation. The examples were selected to illustrate how an IL description can be constructed, and how hand simulation is done. When using a device of any kind, but particularly a computer system, the user has to work with two representations (at least): one of the domain task and one of the device. The user's representation of the domain task will depend on their domain knowledge, and sometime as analysts we have to make informed guesses about the appropriateness of a particular domain representation. Domain knowledge acquisition should ideally be performed to find out what representations are most appropriate, but this is not always possible. The user's representation of the device task can be inferred more directly through studying the device and the way it is presented to the user. One of the important features of PUMA is that it supports the analyst in locating mis-matches between the device representation and the user's conceptual model (which includes both device and domain components).

We have shown how the IL can be used to help the analyst identify mis-matches between the likely conceptual model and the device model, both in terms of the knowledge the user has easy access to and in terms of the ways they are likely to use that knowledge when interacting with the device. We close this section by considering the scope of PUMA (when is it actually worth going to all this effort in a design context?). More substantial examples, which push the approach further, are presented in the following section.

Scope of PUMA

The kinds of insight about usability that a PUM analysis can (and cannot) reveal include the following:

• PUMA can highlight knowledge that users will need that is not readily available to them.

• It can help to identify mismatches between the designer's conceptual model and the way that model is projected to users which may make the device difficult to use effectively.

• PUMA can be used to identify situations in which users might use knowledge inappropriately, or might make rational and reasonable errors as a consequence of the way they apply their knowledge.

• It can help identify inconsistencies in patterns of interaction, or in the ways users are expected to apply their knowledge.

• It is more suited to the analysis of novice, rather than expert, behaviour, but can be applied to either (or simply to users with different background knowledge).

• Similarly, PUMA is better tailored to situations in which users have clearly defined tasks (goals to be achieved), and are expected to know (or be able to identify) actions that lead towards the achievement of those tasks, than to — for example — exploratory or browsing situations in which user goals are less easily specified.

As illustrated in section 2, PUMA can help highlight anomalies in the grouping of functionality (e.g. on buttons or in pull-down menus) on a graphical user interface, but is not particularly well suited to other aspects of graphical interface design; it is best suited to situations where the user's ability to perform tasks depends on her having appropriate knowledge of the domain and device.

Learning to do cognitive modelling takes time and practice. Our aim has been to focus on the basics, so that you understand the principles. Section 2 focuses on how you can apply your understanding in a range of contexts. As expertise develops, however, (over days or weeks, rather than hours), the PUM analyst can learn to rely less on the notation and hand simulation, and more on simply using the underlying concepts as a filter for looking at a proposed design solution. Gradually, PUMA becomes less of a notation and more of a way of thinking. We hope that this first section has given a flavour of what it is to do PUMA, the kinds of representations that are used and the kinds of insights it can help give.

Section 2: Programmable User Modelling Analysis in practice

In section 1, we introduced the basic principles on which PUMA is based. This section is much more concerned with the practicalities of doing PUMA. In this section, we:

- illustrate those practicalities by presenting two large scale examples: a groupware tool and a safety-critical application; and
- work through two smaller examples that are selected to illustrate contrasting points about the approach. These are intended to support the reader gaining direct experience of modelling.

Here we build, through large scale and worked examples, a broader picture of what kinds of issues IL is suited to addressing, and how it can be applied in practice.

7. Large-scale example: a groupware tool

Our first example is taken from an analysis of ECOM, a system developed within the EuroCODE project (Esprit Project 6155). This project was aiming to create an open development environment for computer systems to support people working together over distributed sites. In this case the system being analysed was an early prototype, the product of an exploratory design process, where ideas were being generated rapidly, tested, and refined or abandoned.

The design of ECOM

ECOM is a media-space that enables workers to make connections to colleagues at other sites, to support their collaborative working. It allows individuals to make a variety of audio-visual connections to other individuals or to sites, and to control their accessibility.

The version analysed has its roots in two existing media-space designs, RAVE[13] and CaveCAT[14], and incorporates features of both that were considered desirable. In particular, features of the access control mechanism (by which a user can define which other users can make connections of specified types) are inherited from each of RAVE and CaveCAT, as discussed below.

The ECOM system has a *connect* two-way audio-visual connection, a short one-way *glance* video-only connection, a *snapshot* connection which grabs a still video image from the target user's camera, and the facility to send a *message* to another user (Figure 13). To initiate a connection, the caller selects the target's name from a pull-down menu of users, then a small graphic image of the target (Faber in Figure 13) is displayed next to a door icon. The caller then selects the type of connection required. Depending on the target's access level (to this caller), the connection may be immediately established, established following an acceptance protocol, or refused. The door icon displayed next to the target user shows how that user has set her general accessibility level; this may be open, ajar, closed or barred; in Faber's case, it is closed.

---

[13] See GAVER, B., MORAN, T., MACLEAN, A., LOVSTRAND, L., DOURISH, P., CARTER K. AND BUXTON, B. (1992). Realising a video environment: EuroPARC's RAVE system. In *Proceedings of CHI'92, ACM Conference on Human Factors in Computing Systems*, pp.27-35. ACM Press: New York.

[14] See MANTEI, M., BAECKER, R., SELLEN, A., BUXTON, W., MILLIGAN, T. AND WELLMAN, B. (1991). Experiences in the use of a media space. In *Proceedings of ACM CHI'91: Human Factors in Computing Systems*, pp. 203-208.

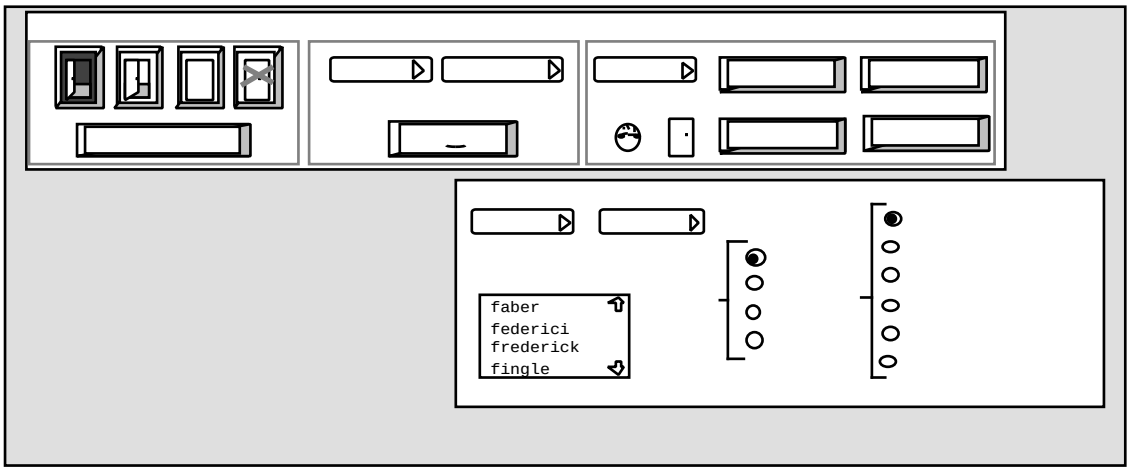


Figure 13: the ECOM interface for making connections and setting exceptions

Each user exercises control over who can access her by setting a general accessibility level (achieved by clicking on one of the door icons on the left at the top of Figure 13). This aspect of the design is inherited from CaveCAT. The system also allows adaptable accessibility to correspond with different interpersonal relations — a feature inherited from RAVE. To set the special permissions which are to correspond with each door state she clicks on the *exceptions* button underneath the door icons. This opens up another dialogue box (the lower window of Figure 13) with two pull-down menu buttons "user ▷" and "group ▷". Selecting from one of these causes the user or group name to appear. The user then selects one of the connection types from the "con" list and one of the condition types from the list on the right (from *always* to *never*).

Three issues were identified by the designers as being ones on which they would value analysis. These issues were expressed in fairly abstract terms — for example "How well does the information displayed by the ECOM interface conform to the underlying system states and potential actions?"

Analysis

As outlined in section 1, the first step in the analysis is to identify candidate tasks, in terms of initial and desired states, and establish the sequence of operations the user would have to perform to effect the state change. Then for each conceptual operation, we ask what its subgoaling preconditions and filtering preconditions are and what its user purpose is — i.e. why the user would want to select this operation.

Identifying appropriate tasks is a matter of predicting which tasks are likely to be most common, according to the available information, and spotting any tasks that might be potential sources of user difficulty. In this case, one obvious task to consider was that of making a connection to another worker. As we worked through this task, it became clear that there was something slightly problematic about the combination of general and specific access control mechanisms, so we paid more attention to that; that, in turn, led us to realise that there were inconsistencies in the way the designers proposed to implement the specific access control mechanism. For a comparatively large system such as ECOM, it is not generally cost-effective to do a complete PUM analysis of all aspects of the design, but to concentrate attention on areas that appear to be problematic in some way.

Constructing an IL description is an iterative process that involves identifying some device objects and relations, defining some operations, adding or modifying device objects and relations, defining the effects of commands on the device state, etc. Here, we try to demonstrate how our analysis derives from the process of constructing the IL description. As is the case for many design and analysis techniques, the main insights are generally gained from going through the process of describing the problem in a particular way, rather than simply from viewing the resulting semi-formal description.

The analysis focuses on what the user needs to know in order to be able to achieve her goals with

the device, and how the device helps provide that knowledge. In particular, we present aspects of the analysis based around the user tasks of making a connection to another worker and setting general accessibility exceptions.

Making a connection to another worker

At its simplest, making a connection to worker T involves identifying T as the target user, then selecting the connection type. We can give outline IL descriptions of these operations:

```
operation identify-target (user: T)
      user-purpose:           target-selected (T)
      filtering-precond:      is-a-registered-user-of-ecom (T)
      action:                 select-target (T)

operation request-connection (user: T, connection-type: C)
      user-purpose:           connection-established (S, T, C)
      subgoaling-precond:     target-selected (T)
      action:                 press-button (C)
```

The first of these operations says that to have the target user specified, the user has to select that target's name, and that this is possible as long as the target's name is in the pull-down list of names in the connection dialogue box. The second states that for a connection of type C to be established between user S and target T, pressing button C will initiate such a connection provided that the target has already been named. This last is a knowledge precondition; performing the same action when the precondition is not satisfied is a legitimate device action, which will connect S to some other, previously specified, user, but it will not address the user purpose. The need for this knowledge precondition leads us to predict that novice users may make order errors, specifying the connection type before the target user.

The definition of the request-connection operation does not say that user S has to know anything at all about the accessibility of the target before requesting the connection. As with normal telephone calls, it is possible to request a connection without having any information about whether or not it will be successful. However, one feature of the proposed design is that the target user's door state is displayed next to her name. The display of the door state conveys information about the accessibility of the target user. If we take account of this, we can express the revised operation:

```
operation request-connection (user: T, connection-type: C)
      user-purpose:           connection-established (S, T, C)
      subgoaling-precond:     target-selected (T)
      filtering-precond:      connection-is-permitted (S, T, C)
      action:                 press-button (C)
```

Here we, as analysts, are looking at the design, and assessing the designer's intention in making particular information available. The next step is to assess whether the information displayed satisfies that intention, and how the user can make use of the information that is displayed. In fact, the door icon indicates the target's current default accessibility, and not her accessibility to this particular user. The door icon is a poor means of representing either the target's actual accessibility or her current openness to being interrupted, and therefore does not serve its (apparent) intended purpose. In part, this problem arises from the incorporation of incompatible mechanisms from RAVE and CaveCAT[15]. By representing the user's task and the information she is intended to use to select her next action, we can identify ways in which the information being displayed is inappropriate.

Setting general accessibility and setting exceptions

The second excerpt from our analysis of ECOM illustrates how PUMA can help identify inconsistencies, clarify the nature of them, and hence guide design modifications.

We start by defining two operations: to set general accessibility and to set an exception. In these

---

[15] This is discussed at more length in BELLOTTI, V., BLANDFORD, A., DUKE, D., MACLEAN, A., MAY, J. & NIGAY, L. (1996) Controlling Accessibility In Computer Mediated Communications: A Systematic Analysis Of The Design Space. *HCI Journal.* **11.4** pp.357-432.

definitions, the term “level” is used to refer to a particular accessibility level (corresponding to a door state), while “at-level” means “when my general accessibility is at this level, or more liberal”.

```
operation set-accessibility-level (level: L)
      user-purpose:           has-general-accessibility (S, L)
      subgoaling-precond:     is-logged-in (S)
      action:                 click-on-door (L)

operation set-individual-accessibility (user: U, connection-type: C, at-level: L)
      user-purpose:           has-specific-accessibility (S, U, C, L)
      subgoaling-precond:     is-logged-in (S) &
                              user-named (U) &
                              connection-type-specified (C)
      action:                 press-radio-button (L)
```

These operations make use of several object types and relations:

```
OBJECTS:
      user: S, faber, federici, frederick, fingle,...
      connection-type:  connect, glance, snapshot, message
      level:      open, ajar, closed, barred
      at-level:   always, open, ajar, closed, barred, never
RELATIONS:
      has-general-accessibility (user, level)
      has-specific-accessibility (user, user, connection-type, level)
      user-named (user)
      is-logged-in (user)
      connection-type-specified (connection-type)
```

Most of this definition is unproblematic. However, it has been necessary to introduce two terms, “level” and “at-level”, that have related, but different, meanings. The first of these is represented at the interface by the doors, and the other by the radio buttons. To understand the effects of setting exceptions, the user has to understand the relationship between these two terms. The meaning of “at-level” is not clearly conveyed by the current display because radio buttons do not clearly represent cumulative effect. One of the consequences of this poor mapping between display and meaning is that it was easy for the designer to get “always” and “never” in the wrong order. “Always”, as the most liberal setting, should be next to “[even] when door locked”, while “never”, as the most restrictive, should be next to “[only] if door open”. This fault had passed unnoticed by several other people before being recognised while defining what the interface objects were intended to mean to the user of the device. This is an example of a design issue where PUMA helped to identify aspects of the interface design for which a closer mapping between the surface representation and the underlying system variables would have made the interface much easier to use.

Referring back to figure 13, we note one further point. The grouping of the four doors and the “exceptions” button in one box on the display suggests that they are semantically grouped — that they serve similar purposes. As the analysis above demonstrates, they serve quite different, but superficially confusable, roles (with a selected door icon defining the user’s current accessibility, and the exceptions button allowing the user to change the meaning of each door icon for particular colleagues); their grouping is likely to exacerbate the user’s confusion. Here we have two examples (in the criticism of the use of radio buttons and of the grouping of objects on the display) where PUMA can yield insights regarding aspects of the display design, although the technique does not in general address questions of visual structure.

About this analysis

As demonstrated for this exemplar, a knowledge analysis can be done for particular issues long before the design has been specified to a level of detail where it can be used to build a runnable model. The process of representing the design in the IL can highlight inconsistencies, hidden assumptions and unresolved design issues, which can inform the next cycle of refinement and modification in the design process. Also, the analysis can show *why* particular aspects of the design are likely to be difficult to use, which should help to guide the re-design.

8. Worked example: Identifying incompatibilities between the system model and the user's model of the system[16]

The next example illustrates how insights can be achieved without even doing extensive hand-simulation. It is intended to give the reader the opportunity to work through an analysis in detail.

One role for the Instruction Language is to help the analyst identify incompatibilities between the system model and the user's model of the system. Such incompatibilities are often caused by inconsistencies between the internal device state and the rendering (or displayed representation) of that state.

We take as our example the relationship between the locations of files on discs and on the Macintosh desktop. One feature of the system design is that files can be moved around within the file structure on one disc by dragging their icons; however, when transferring files between discs, the effect of the drag operation is to copy the file. To copy a file to a different folder on the same disc, the user has to press the "option" key while dragging the file icon. As an isolated design decision, this implementation of the effect of dragging a file icon is based on a clear rationale — that one generally needs only one copy of a file on a disc, so a different, distinctive action has to be performed to make a second copy on the same disc. In contrast, if a file is transferred from one disc to another, the user is likely to want to keep both copies: maybe one is a backup of the other. A second feature of the system design is that files from either disc can be located temporarily on the "desktop" (see figure 14). Again, as a design decision, this seems reasonable; a temporary storage place (or a place to keep frequently accessed files) has clear uses. However, it is possible that these design features might interfere with each other. We generate a scenario of use that will allow us to investigate this aspect of the design — for example, considering the implications of having files from two different discs stored on the desktop, and analysing what the user will need to know to be able to move or copy those files around. We take as our example a scenario in which the file *contacts* has, at some previous time, been moved from a folder on the hard disc, and the file *addresses* from a folder on the floppy disc, and both are resident on the desktop. Now, the user wants to have copies of both files in the assay-demo folder, which resides on the hard disc, so both files are to be copied there.

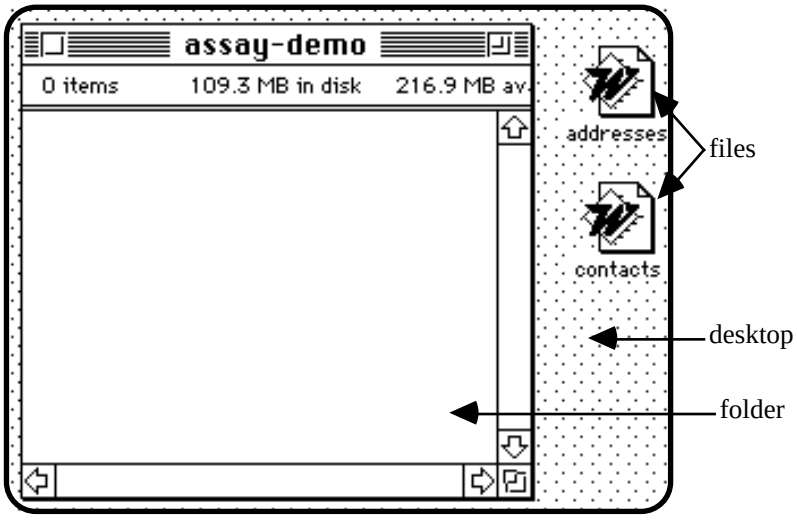


Figure 14: the Macintosh™ desktop

> Produce an IL description for this scenario. The representation should show that the user knows about files, folders, the desktop and discs, that every file and every folder is stored on a disc, and that files may be located in folders or on the desktop.
>
> However, you should aim to keep the representation **simple**; ignore difficult representational issues such as the nature of a "copy" of a file.
>
> Use your description to discuss usability problems with this device.

Our IL description and usability analysis is shown on the next page; compare yours with it.

---

[16] This example and the one in Chapter 10 are adapted from BLANDFORD, A. E. & YOUNG, R. M. (1996) Specifying user knowledge for the design of interactive systems. *Software Engineering Journal*. 11.6, 323-333.

Our analysis is as follows. We say that the user knows about files, discs and folders, that she knows that files and folders are stored on discs, and that files can be in folders, or on the desktop.

```
OBJECTS
file: addresses, contacts
disc: hard, floppy
folder: assay-demo

RELATIONS
file-on-disc(file, disc)      - (cannot be seen directly)
folder-on-disc (folder, disc) - (cannot be seen directly)
file-in-folder(file, folder)  - detected-by-looking
file-on-desktop(file)         - detected-by-looking
```

From a user's point of view, what he wants to do is to copy a file, so we define conceptual operations to copy files. Given the 'special' nature of the desktop, we define special operations that only apply to copying from the desktop (rather than more powerful operations that would also apply to copying from one folder to another). While the computer scientist's instinct might be to define more general operations, we do not demand this sophistication of the user's knowledge! For completeness, we also define the conceptual operation for moving a file from the desktop to a folder within a disc (though this is not strictly necessary for this analysis). The operations we define are those for a reasonably sophisticated user of the system who understands the significance of the filtering precondition— that dragging has different effects depending on whether the file and its destination folder are on the same or different discs, and that he has to press "option" while dragging in order to copy files within one disc.

```
OPERATIONS
operation copy-onto-same-disc (file:F, folder:R)
  user-purpose:         file-in-folder(F, R) &
                        file-on-desktop(F).
  filtering-precond:    there is a D such that
                        file-on-disc(F, D) & folder-on-disc(R, D)
  action:               option-drag-file(F, R).

operation copy-onto-other-disc (file:F, folder:R)
  user-purpose:         file-in-folder(F, R) &
                        file-on-desktop(F).
  filtering-precond:    there is no D such that
                        file-on-disc(F, D) & folder-on-disc(R, D)
  action:               drag-file(F, R).

operation move-file (file:F, folder:R)
   user-purpose:        file-in-folder(F, R) &
                        not file-on-desktop(F).
   filtering-precond:   there is a D such that
                        file-on-disc(F, D) & folder-on-disc(R, D)
   action:              drag-file(F, R).
```

In this case, we are saying that the things the user knows about the device state are only what they can find out from the display. Also that the desired state is one in which both files are copied to the assay-demo folder.

```
USER KNOWS INITIALLY

DESIRED-STATE
file-in-folder(contacts, assay-demo)
file-in-folder(addresses, assay-demo)
file-on-desktop(contacts)
file-on-desktop(addresses)
```

Finally, we need to define the effects of user actions on the device, the initial device state, and what is displayed.

```
DEVICE COMMANDS
drag-file (file:F, folder:Destination)
This command puts file F into the destination folder.  If the
file is already on the same disc as the folder, this is treated
as a Move operation.  But if they're on different discs, then it
is treated as a Copy: a copy of the file is put in the folder.

option-drag-file (file:F, folder:Destination)
This command copies file F into the destination folder.

INITIAL
folder-on-disc(assay-demo, hard).
file-on-desktop(contacts).
file-on-desktop(addresses).
file-on-disc(contacts, hard).
file-on-disc(addresses, floppy).

DISPLAYED
file-in-folder(file, folder).
Provided that all kinds of conditions are met (folder is open,
scrolling is such that the file is within the boundaries of the
folder-window, the relevant part of the folder-window is on
screen, and is not obscured by another window) … then the screen
shows that a file is in the folder.

file-on-desktop(file).
      Similarly.
```

It would also be possible to write an IL description for the novice user who does not know about this filtering precondition; in this case the IL description would show that the device behaviour is unpredictable to the user, and that the user would be unable to copy files within one disc.

The process of doing this analysis, and the resulting IL description, highlight a problem. Because we do not define the user's initial knowledge of the device state, as it should be acquired by looking at the display, the knowledge-condition expressed in the filtering precondition cannot be checked by the user.

According to this analysis, there are two obvious problems:

(a) When about to do the copy-file operation on one of the files, the user checks the filtering precondition condition, and realises that he doesn't know which operation has its filtering precondition satisfied, and the information is not visible.

(b) If he doesn't actually *check* the condition, but instead assumes that the files are on a different disc from the folder, then when he drags the "contacts" file to the folder he will be surprised by the fact that it is moved, rather than being copied.

The obvious way to change the design so that these problems are avoided is to assign the desktop to the start-up disc (i.e. the disc the system cannot do without), and to arrange that files stored on other discs can appear on the desktop only if copied there. (That is what is already done for "remote" files mounted across a network.) However, it should be noted that although this solution addresses this particular problem well, it does not address the more general problem that there are often situations in which the user cannot easily tell which disc a particular folder is stored on.

The analysis given here is as simple as possible. For example, it does not attempt to represent the notion of a "copy" of a file: that the copy has (initially) the same name, that it has the same contents, that for some purposes it is equivalent to the original, but that the copy and the original can be changed independently and so become out of step. Even so, it highlights a source of user difficulty when using the system.

This example illustrates the fact that some designs that are clear and rational from a device perspective actually hide essential information from users as a consequence of inconsistencies between the device state and the way it is presented to the user. A user-centred IL analysis can help to clarify the knowledge needed by the user to work effectively with the device, and thus

highlight inconsistencies between the device state and the user's knowledge of that state.

9. Large-scale example: a high-reliability system

The next analysis we present relates to the CERD (Computer Entry and Readout Device), a component of an Air Traffic Control Officer's workstation. In this case, the design was fairly mature; it was also the product of a formal design process, in which the design team was using formal specification techniques, and occasionally executing proofs for the most safety-critical aspects of the design.

Introduction to the CERD exemplar

The CERD is a device that allows an Air Traffic Control Officer to access and manipulate the landing order of aircraft at a major airport complex. The description that follows is simplified, but includes all the detail relevant to this analysis.

The system is used to control the order in which aircraft will be permitted to land, by allowing the Air Traffic Control Officer to construct messages and send them to the National Airspace System. Such messages can request changes to the landing order or the assignment of flights to special categories. The CERD also receives messages from the National Airspace System and displays them so that the user can read and respond to them if necessary. Figure 15 shows a typical CERD display, including the message line (on which messages from the National Airspace System are displayed), the scribble line (on which the user constructs messages to send) and flight keys which allow the user to refer to individual flights in a message.

The Air Traffic Control Officer constructs a message by pressing keys on the CERD that correspond to components of the desired message. For example, a message requesting that the order of flights BM11 and AL3233 be swapped would be constructed by pressing the keys corresponding to "*swap*", "*bm11*", "*al3233*" and "*confirm*", while assigning an "emergency" code to flight BM11 involves pressing "*assign*", "*bm11*", "*em*" and "*confirm*" ("*em*" being available from a different screen, which is displayed when appropriate). Only keys that can validly be pressed next are enabled. All requests constructed by an Air Traffic Control Officer are sent to the National Airspace System, and may be either accepted or rejected. There are four request types: to assign a special category indicator (*Assign*); to reposition one flight in the landing order (*Repos*); to swap two flights (*Swap*); and to resequence a block of up to 10 flights (*Reseq*).

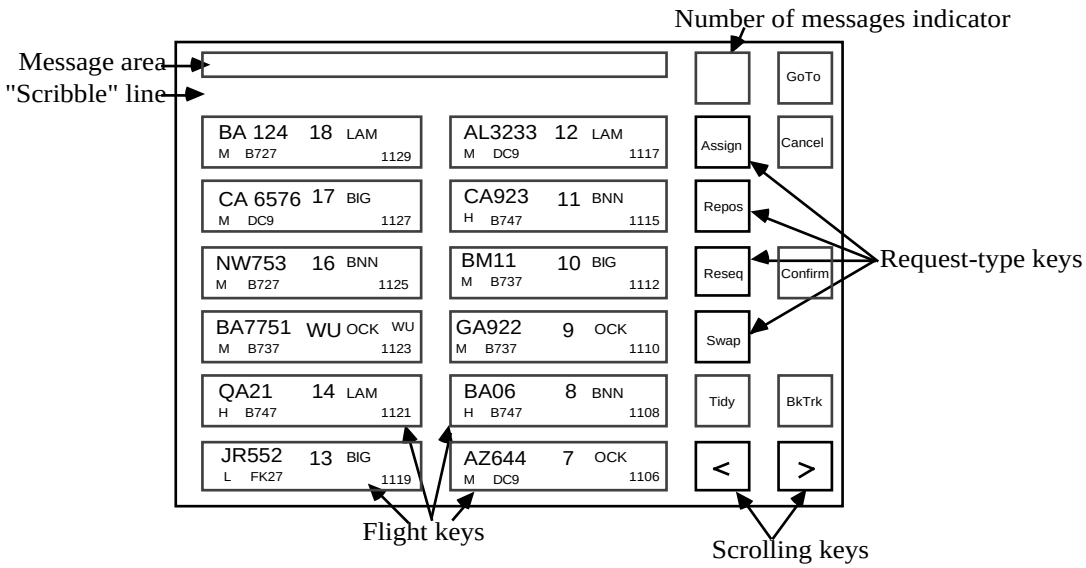


Figure 15: Typical structure of a CERD page (this display corresponds to columns 1 & 2 of Figure 16 at time T1)

The flight keys are designed to be similar to the paper flight strips that are traditionally used to monitor flights. They display the flight number, the type of aircraft, the estimated landing time, etc. The flights area of the CERD display is a moving window, which is controlled by the

scrolling keys, giving the Air Traffic Control Officer information about up to 12 consecutive aircraft in the landing order.

If the current message on the top line refers to a particular flight then the *GoTo* key becomes enabled, and pressing the key will cause the CERD to display, in a tidy state, the screen containing information about that flight.

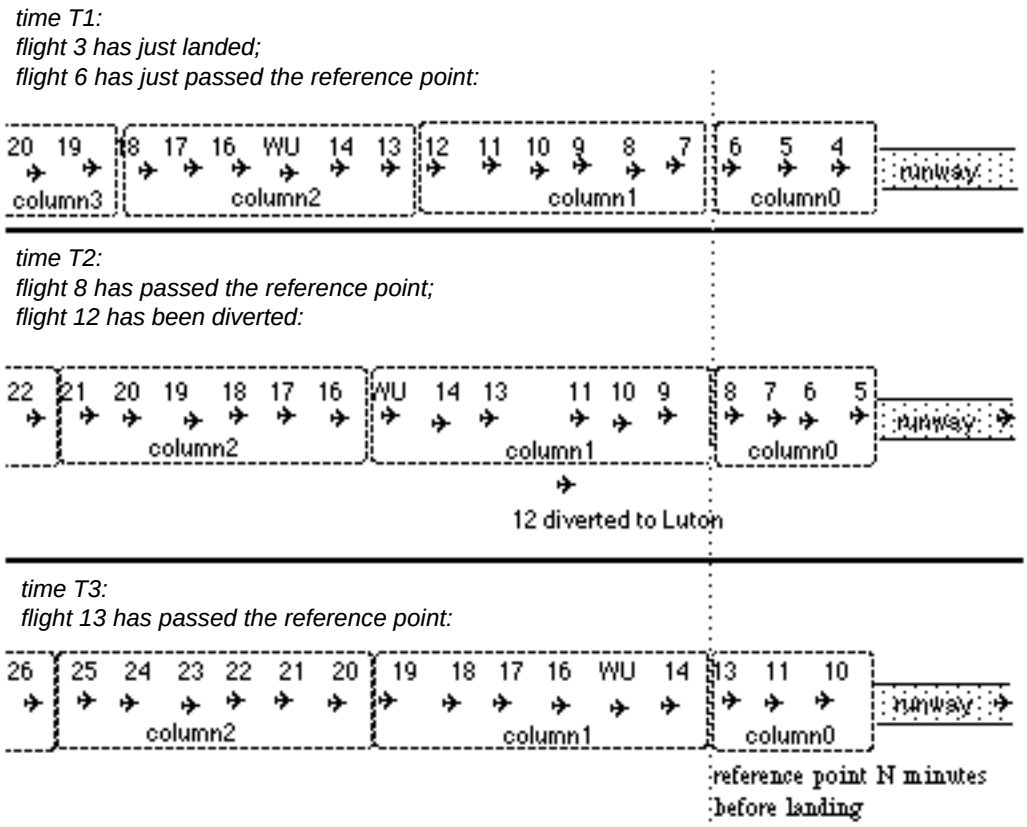


Figure 16: six flights are eligible for display in each column of a CERD page
(the display will show two adjacent columns, counting forward or back from the reference point).

At any instant, a flight is eligible for display in a particular column, as illustrated in Figure 16; flights become eligible to appear in a new column as other flights pass the reference point, or get diverted. If the flight becomes ineligible for display in either of the columns on the screen (due to its being diverted or because earlier flights have passed the reference point, for example), then it disappears from the display. Typically, this means that slots at the bottom right are cleared as flights become eligible for display in the next column. There is an explicit *Tidy* control that becomes available once the display is untidy; this updates the display by removing intervening gaps. If there are more flights that can be displayed then one or both of the scrolling arrow keys (Figure 15) are highlighted. Scrolling is done on a column by column basis. Scrolling causes the flights in the next block to be displayed in a tidy way.

Here we present extracts from our analysis of just one of the issues we were asked to address: "Are CERD's current operations the best ones to cover the tasks which need to be performed? Could they be better supported by another set of operations?"

Design issue: tasks and CERD operations

Again, the first step in the analysis is to identify candidate tasks, in terms of initial and desired states, and establish the sequence of operations the user would have to perform to effect the state change. Here, we consider just two such tasks.

First task: checking the landing order

One task Air Traffic Control Officers perform routinely is to work through the landing order from front (nearest to landing) to back (furthest away), checking that it is satisfactory. For example, if a large aeroplane is immediately followed by a very small one then extra time has to be allowed between landings (to allow for turbulence), so they will try to re-order the flights to avoid this. The task involves the Air Traffic Control Officer scrolling left through the flights in the landing order, checking flights, and occasionally constructing messages to request changes. We focus on the scrolling operation, which has the purpose for the user of seeing the next flights in the landing order. The operation is only possible if there are further flights to be displayed.

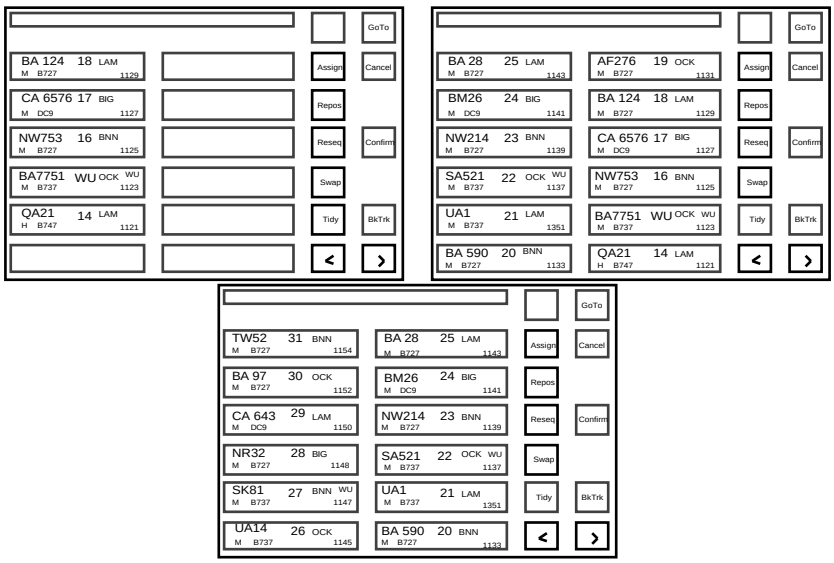
We can construct a semi-formal description of this operation:

```
operation see-next-set-of-flights-left
      user-purpose:           see-next-flights-further-away
      filtering-precond:      there-are-more-flights-left
      action:                 scroll-left
```

The corresponding device command is:

```
device-command scroll-left
      effect:     tidy-screen; then
                  display-next-block-of-flights-to-left
```

One aspect of analysis is to consider whether the effect matches the user purpose, to ascertain whether the effect is predictable to the user, and whether there are circumstances in which there are potential mismatches between the user's intention and the device effect. In the case of scrolling to see more flights further away, it clearly depends on what the effect of the system *Tidy* is. If the screen is very untidy just before the user applies this operation then flights might be substantially re-organised on the display. In particular, if seven flights have landed or been diverted then it is possible for flights to jump from "off to the left" to "off to the right" with one scrolling operation (Figure 17), so that the user-purpose of seeing the next flight to the left (in this case, flight 19) is not satisfied. Here, the role of the modelling is to identify circumstances in which the device effect fails to match the user purpose.



(a) (b) (c)

Figure 17:
(a) original screen at time T3 (Fig. 16) — 7 flights have landed or been diverted since the screen was last tidied.
(b) the screen as it would appear if the user pressed the "tidy" button
(c) the screen as it appears if the user presses [<] ; flight 19 has jumped from left to right.

As user modellers, we cannot state definitively whether or not this mismatch is actually likely to cause difficulties. However, the results of this analysis can be used to frame a focused question to a domain expert ("Is this likely to happen, and what might the consequences be?") or to identify a point that needs to be brought out in training (so that users are trained to recognise and respond appropriately to this situation).

This first analysis is of a task that we were explicitly told about by the CERD designers. We have not considered the details of message construction here, but identified a potential problem with the implementation of scrolling. We now move on to consider various aspects of message construction.

Second task: sending two messages

The second task is more detailed: we consider how the Air Traffic Control Officer might interleave two message-sending tasks, one based on the currently visible screen and one in response to a message that has just appeared on the message line. For example, we might imagine that the situation in the air is as shown in Figure 16 (time T2), but that the display, showing columns 1 and 2, has not been updated since time T1, and is as shown in Figure 18. The Air Traffic Control Officer has decided to swap the flights either side of the diverted one, and a message arrives to say that flight BA772 (which is not currently displayed) should be given an "emergency" special category indicator. The likelihood of such a situation arising should be assessed by domain experts. The conclusions we draw are made subject to our assumptions being valid, and serve as a focus for asking motivated questions about relevant aspects of the domain context.

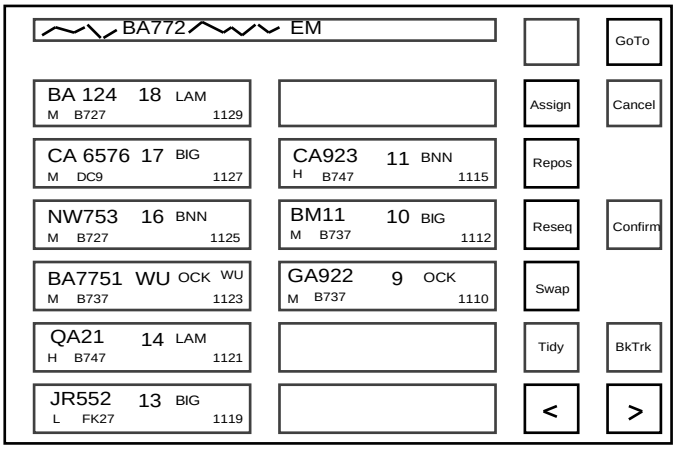


Figure 18: example initial state

The task consists of two sub-tasks — constructing and sending a message to change the special category indicator of BA772, and constructing and sending a message to swap CA923 and JR552.

The operations needed for constructing a message about a flight are to specify the request-type, specify the flight (which has to be on the display), construct the remainder of the message, and then send the request. We need to be able to talk about flights, messages and the words that make up messages, and about relationships between these objects:

```
OBJECTS
flight:     ba772, bm11, ca923, jr552, qa21...
message :   m1, m2 ...
word: assign, repos, ba772, bm11, em...

RELATIONS
shown-on-screen (flight)
references-flight (message, flight)
sent-message (message)
constructed-message (message)
next-in-request (word)
current-message(message)
```

In this scenario, the user's task is to be in a state where the two messages have been sent:

```
desired-state(sent-message("assign BA772 em"),
              sent-message("swap CA923 JR552"))
```

We start by considering message construction. To construct a message, the user selects a message-type key, specifies the first flight, then constructs the remainder of the message (we do not consider the details here), and finally sends it by pressing *"confirm"*.

For specifying the message type, the subgoaling precondition is that no request-type has already been selected, and the user purpose is that the message type appears as the next item in the message being constructed on the scribble line. There are additional device effects (e.g. the display changes with some previously greyed out buttons becoming activated and some previously activated ones being greyed out). The user can rely on the display state to ascertain what actions are valid at any given time.

```
operation add-message-type-to-request(message-type: T)
      user-purpose:         next-in-request (T)
      subgoaling-precond:   (there is no partially constructed message)
      action:               press-T-key
```

The first flight in the message can be specified when the message-type has been entered, and the flight key is displayed on the screen:

```
operation add-first-flight-to-request(flight: F)
      user-purpose:           next-in-request (F)
      subgoaling-precond:     next-in-request(T)
                              shown-on-screen (F)
      filtering-precond:      (T is a message type)
      action:                 press-F-key
```

We will gloss over the construction of the remainder of the message with a high-level operation. The reason for splitting message construction into "message type", "first flight" and "the rest" will become clear below:

```
operation construct-rest-of-message(message-type: T, flight: F)
      user-purpose:         constructed-message(T,F,<rest-of-message>)
      subgoaling-precond:   next-in-request(F)
      action:               <press appropriate keys>
```

Finally, the user must send the message:

```
operation send-message(message: M)
      user-purpose:           sent-message(M)
      subgoaling-precond:     constructed-message(M)
      action:                 press-confirm
```

Without presenting a full hand-simulation here, we can reason as follows. The user has two goals: to have sent each of the messages. The user has to select which to address first; he might choose the first because it is an emergency, or he might choose the second because all the necessary information is available on the current screen. At this point we have a non-deterministic choice, and we can explore the consequences of each possibility, but we consider just one case: sending the "emergency" message first. To have sent this message, the user selects an operation with the desired user-purpose. According to the IL description, the user knows that the appropriate operation is to send-message; this involves pressing "confirm" after the message has been constructed; therefore, he acquires the new goal of having the message constructed (i.e. displayed completely and correctly on the "scribble" line of the display). This in turn is achieved by typing the rest of the message once the first flight has been specified. For specifying the first flight, there are two subgoaling preconditions: that the message type has been specified, and that the flight is displayed on the screen. The user adopts both of these as goals, and might address them in either order (according to the current IL definition). The first of these is addressed by applying the operation to press the message-type ("*Assign*") key, and the second by pressing *GoTo*. We have not yet specified the *GoTo* operation. For *GoTo*, the filtering precondition is that a flight has been mentioned in the current message, and the user purpose of the operation is that the flight is displayed ("shown-on-screen"). The *GoTo* operation is only available if there is a current message which references a flight; then the effect of *GoTo* is that the referenced flight appears on the screen. We say that the purpose of applying the operation is to make the flight available for

selection (i.e. shown on the screen):

```
operation find-flight-from-message (flight: Fl)
      user-purpose:        shown-on-screen (Fl)
      filtering-precond:   references-flight (current-message(), Fl)
      action:              goto
```

As stated, this operation seems unproblematic; it enables the user to easily locate a flight that is needed for the current task.

If the user addresses the goal of adding the message type to the request before that of getting the flight displayed on the screen then he becomes committed to applying the operation add-message-type-to-request("assign"); since the precondition is (we shall assume) satisfied, the user can press the corresponding key. The device state is updated accordingly. There is still a subgoaling precondition on add-first-flight-to-request("BA772"), that the flight is displayed, so the user addresses this goal, which results in him pressing the *GoTo* key. However, pressing *GoTo* clears any partially completed message on the scribble line:

```
device-command goto
      effect:     clear-scribble-line; then
                  tidy-screen; then
                  display-block-of-flights including referenced flight
```

Consequently, if the user presses *Assign* and then *GoTo*, the effect of pressing *Assign* will be undone and the user will have to re-adopt the goal of adding "Assign" to the message. Although both are used for navigating, *GoTo* and scrolling differ in their effects.

Unless primed with the (over-generalised) knowledge that the flight must be displayed before pressing the assign key, hand simulation highlights the possibility of order errors, pressing *Assign* before *GoTo*. Since it is not obvious that the flight has to be visible before the user presses *Assign*, and is not true if the user is finding the flight by scrolling, this leads us to predict that users may make order errors. This prediction depends on two things:

- Firstly, it depends on us, as analysts, not specifying as a subgoaling precondition that the flight should be on the screen before the request-type operations are performed. This is consistent with the heuristic noted above, that the designer should assume the minimum possible level of user knowledge and explore the consequences of that, in order to understand better what the knowledge requirements on the user are.
- Secondly, it depends on the ability to hand-simulate the behaviour of the model.

Here, the role of the user modelling is to identify a potential usability problem and highlight an inconsistency in the implementation of related operations.

Now we consider re-locating the display relative to the landing order. Having sent the "assign emergency code" request the user now has to construct the "swap these flights" request. Here, we just consider the sub-task of getting the flights on the display. To do this, he will need to return to the screen that was displayed before he responded to the "emergency" message using the scrolling keys. For the scroll-left device command this is a different user purpose from that in the previous (checking landing order) task, so we construct a different conceptual operation. The user has various cues about his current position and previous position in relation to the landing order, but they all require significant mental processing. If we try to describe the finding-flight operations, we need a knowledge filter that is difficult to express precisely:

```
operation find-flight-later-in-landing-order (flight: Fl)
      user-purpose:        shown-on-screen (Fl)
      filtering-precond:   is-later-than-currently-displayed (Fl)
      action:              scroll-left

operation find-flight-earlier-in-landing-order (flight: Fl)
      user-purpose:        shown-on-screen (Fl)
      filtering-precond:   is-earlier-than-currently-displayed (Fl)
      action:              scroll-right
```

These operations together express the fact that the user needs to know which way to scroll. This knowledge may be readily accessible as part of an experienced Air Traffic Control Officer's mental model of the current landing order; alternatively, it might be available in some form on another display. It is not readily available on the CERD, so a question to the designers would be: where are users expected to obtain this information? The role of the user modelling here is to highlight

the fact that this operation makes demands on the user's knowledge, and to query the potential sources of that knowledge.

In this case, the effects of scrolling do not directly match the user purpose; the user may have to scroll through an unspecified number of columns to locate the desired flights. The Air Traffic Control Officer, as a domain expert, may have a mental representation which will support him well in re-locating the flights, but there is no simple corresponding device command. Here the role of the user modelling is to identify a functional requirement on the design, on the basis of an analysis of the users' tasks.

About this analysis

This analysis was done on a much more mature product than the ECOM analysis. The usability difficulties identified are comparatively minor, though they could have serious implications in a safety critical context. One of the difficulties we were faced with when first doing this analysis what that we are not domain experts, and making informed assumptions about the way air traffic controllers think about their domain tasks was difficult. We may have made some inappropriate assumptions; however, in the analysis we have been forced to make those assumptions explicit, and can use them to guide discussions with domain experts about the validity of the analysis. In an ideal world, designers and analysts would have easy access to domain experts in order to validate assumptions; in practice, such access is often very limited, and should be used for dealing with design issues in a focused way. A PUM analysis can contribute to that.

10. Worked example: Invoking principles of user cognition

If we apply principles about ways in which the user applies her knowledge in performing tasks, we can identify further problems that go beyond the simple "mismatch" problems illustrated in the "desktop" example (section 8). We have already worked through examples that include hand-simulation; this example takes that idea further.

As described earlier, the simplest modelled behaviour is a means-ends analysis schema that takes account of changes in the device state. The user identifies operations to perform on the basis of "selection by purpose" — by identifying operations whose user purpose matches the current goal and whose filtering preconditions are satisfied. If any subgoaling preconditions are not satisfied, they are adopted as subgoals. When an operation with satisfied preconditions is selected, the user issues the corresponding device command and her knowledge of the device state changes in response to predicted effects and perceived changes to the device state. This particular example hinges on the way the user keeps track of the state of the device.

The example is taken from the design of a mail tool that allows users to store messages in named folders. At any time, some folder has been "loaded" into the tool, and the headers of the messages it contains are displayed in a scrollable list. The mail tool provides two operations: to Load a folder of messages the user wants to see, and to Move a message from the currently loaded folder to some other folder. The tool has a slot named *Folder* which can hold the name of a single folder. The names of all existing folders can be found on a menu that the user can see by pressing on either of the operation buttons, *Load* or *Move* (see Figure 19). There are two different ways to invoke the Load and Move operations, both requiring the name of a folder other than the one that is currently loaded. In the case of Load, it is the name of another folder to be loaded. In the case of Move, it is the name of a folder where the user wants a message (from the currently loaded folder) moved to.

- If the folder name is already in the *Folder* slot, then the user can simply click on the *Load* button to load the folder, or on the *Move* button to move the selected message to the folder. (To "click" means to press the mouse button and then immediately release it.)
- Otherwise — if the desired folder name is not in the *Folder* slot — the user must press the mouse button (and hold it down) on the *Load* or *Move* button in order to bring up the menu of folder names, drag the mouse until it is over the intended folder (e.g. "folder5" in the diagram), and then release the mouse in order to invoke the operation with that selected folder. Doing this causes the name of the selected folder to be put in the *Folder* slot.

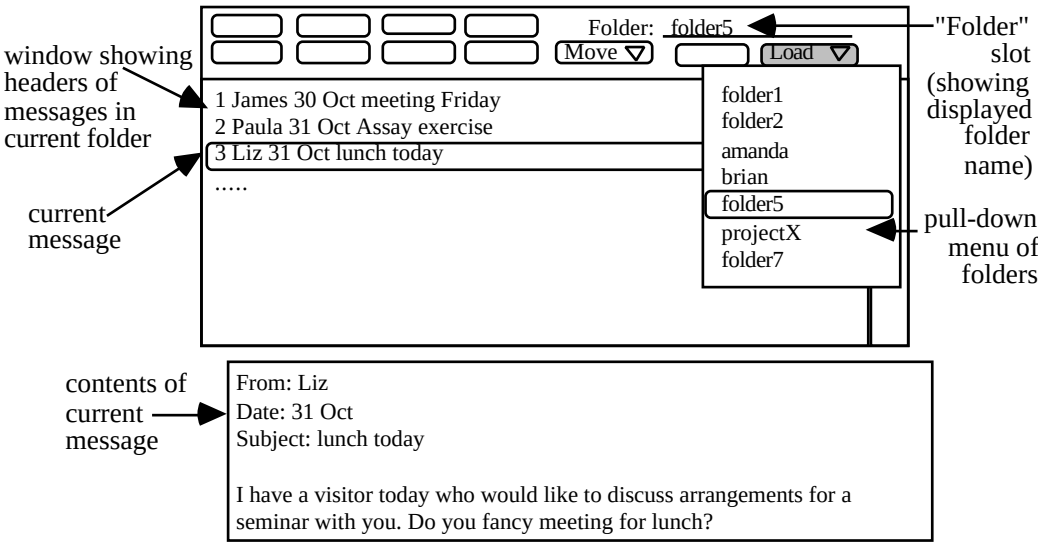


Figure 19: example screen (schematic) from the Mailtool

We consider one scenario of use. The user has several mail files, some of which are sorted by the name of a person they concern, and some by the name of the project they deal with. There are mail messages from Amanda and Brian stored in folders under their names. The user has recently had exchanges with both people about a new project, Project X. The user has created a "projectX"

mail folder, and now needs to go through both the amanda and brian folders, in order to move any messages concerning Project X to the projectX folder.

> Write an IL description to describe this scenario. Use your description to reason about usability of the device. Think in particular about two things:
>
> • from a user's perspective, the task is to move files *from* one folder *to* another, so your representation should include different conceptual objects to represent "from" (or "source") and "to" (or "target") folders.
>
> • the user can get their information about the state of the device from two sources: from the display, or by predicting the main effects of predictable operations. First write your IL description assuming the user will get their information from the display. Then revise it for the more experienced user who relies on keeping track of the device state by predicting the effects of actions.

Our IL analysis for this scenario is shown on the next page.

In our analysis, we have chosen a representation that distinguishes between the selected-from and selected-to folders to reflect their different conceptual roles (as the folders a message is moved from and to respectively). This is an example of a general heuristic that items that serve different roles in the interaction should be distinguished; the consequences of this will become clear in the analysis below. The two relations selected-to and selected-from both involve the user looking at the same place on the screen to get the value of displayed-folder-name. For the user to know which folder is currently opened, we also have a relation *opened*, which is defined as being predicted by the user when a folder is loaded.

```
OBJECTS:
message:    a1, ..., a5, b1, ..., b6
folder:     amanda, brian, projectX

RELATIONS
highlighted(message)          -- detected-by-looking
displayed-folder-name(folder) -- detected-by-looking
opened(folder)
selected-to(folder)           -- selected-to(F) if displayed-folder-name(F)
selected-from(folder)         -- selected-from(F) if displayed-folder-name(F)
visible(message)              -- detected-by-looking.
contains(folder,message)      -- contains(F,M) if opened(F) & visible(M)
OPERATIONS
operation move-quick (folder:Tf, message:M)
      user-purpose:           contains(Tf,M)
      subgoaling-precond:     highlighted(M)
      filtering-precond:      selected-to(Tf)
      predicted-effect:       contains(Tf,M)
      action:                 click-move

operation move-slow (folder:Tf, message:M)
      user-purpose:           contains(Tf,M)
      subgoaling-precond:     highlighted(M)
      filtering-precond:      not selected-to(Tf)
      predicted-effect:       contains(Tf,M)
      action:                 select-and-move(Tf)

operation load-quick (folder:Ff)
      user-purpose:           opened(Ff)
      filtering-precond:      selected-from(Ff)
      action:                 click-load

operation load-slow (folder:Ff)
      user-purpose:           opened(Ff)
      filtering-precond:      not selected-from(Ff)
      predicted-effect:       opened(Ff)
      action:                 select-and-load(Ff)

operation highlight-message (folder:Ff, message:M)
      user-purpose:           highlighted(M)
      subgoaling-precond:     opened(Ff)
      filtering-precond:      contains(Ff,M)
      action:                 pick(M)

USER KNOWS INITIALLY
contains(amanda,a1)...contains(amanda,a5)
contains(brian,b1)...contains(brian,b6)
opened(amanda)

DESIRED-STATE
contains(projectX,a1)...contains(projectX,a5)
contains(projectX,b1)...contains(projectX,b6)
```

```
DEVICE COMMANDS
click-move
      moves the highlighted message to the currently selected
folder.

select-and-move(Tf)
    sets the selected folder to Tf and moves the highlighted
message there.

click-load
      opens the currently selected folder

select-and-load(Ff)
      sets the selected folder to Ff and opens it.

pick(M)
      causes message M to be highlighted

INITIAL DEVICE STATE
contains(amanda,a1)...contains(amanda,a5)
contains(brian,b1)...contains(brian,b6)
opened(amanda)

DISPLAYED
There will be a displayed-folder-name, visible messages, and one
highlighted message (ignoring the case where the currently opened
folder is empty).
```

We have defined conceptual operations to correspond to the device commands that are available to the user. Move-quick involves just clicking the move button; the highlighted message then gets moved to the currently selected folder. In contrast, move-slow involves pressing the move button and selecting the name of the target folder; the highlighted message then gets moved to the target folder, which becomes the currently selected-to folder and the current displayed-folder-name. Similarly, load-quick loads the currently selected-from folder, while load-slow involves pressing the load button and selecting the name of the target folder, which is then loaded and becomes the currently selected-from folder and the current displayed-folder-name. Our final operation, highlight-message, involves clicking on a message header (from the currently opened folder).

As shown in the IL description, this task has some small complexities. Firstly, there are two ways to achieve particular goals, which the user would select in different circumstances (as specified in the filtering preconditions). Secondly, there is only one location on the display that contains the name of a folder; this might be the folder that is currently loaded, or the one to which messages are being moved. This is a case where one display item serves two purposes.

Applying the principles stated above to this IL description, we get predicted behaviour as follows:

- the modelled user has goals as specified in the task.
- applying selection-by-purpose, the user selects operation `move-slow(projectX,a1)`, but its subgoaling precondition is not satisfied.
- applying precondition-subgoaling, the user adopts the goal `highlighted(a1)`.
- the preconditions on the corresponding operation, `highlight-message(amanda,a1)`, are satisfied, so the user does `pick(a1)`.
- the display state changes, so the user knows that this is highlighted, satisfying the precondition for `move-slow(projectX,a1)`, so the user does `select-and-move(projectX)`.
- the user predicts the intended effect of `move-slow`, `contains(projectX,a1)`. The display state changes so that the user also knows `displayed-folder-name(projectX)`; `selected-to(projectX)`; `selected-from(projectX)`.

By the same reasoning, the user then selects-by-purpose `move-quick(projectX,a2)`, and the modelled behaviour continues in this way until the point where all the messages from amanda

have been moved, and it is time to move messages from brian.

The skilled analyst would focus on the section of this analysis shown in the table below — i.e. the transition point where the "from" folder is being changed. As with any design or analysis activity, identifying the areas that merit most attention in this way is largely a matter of craft skill and experience. In this case, as the analysis shows, if the user always depends on the screen for information on the selected folder, then she will not experience problems. This can be contrasted with the variant on this analysis that we present below.

Committed operations & goals, and commands issued	User's knowledge of state

...we get to a point where....
 	
Goals:
contains(projectX,b1)...
  contains(projectX,b6)
	contains(projectX,a1)...
	  contains(projectX,a5)
	contains(brian,b1)...contains(brian,b6)
	opened(amanda)
	highlighted(some-message)
	displayed-folder-name(projectX)
	selected-to(projectX)
	selected-from(projectX)

Selection-by-purpose ==> Operation: move-quick(projectX,b1)	
Precondition-subgoaling ==> Goal: highlighted(b1)	
Selection-by-purpose ==> Op'n: highlight-message(brian,b1)	
Precondition-subgoaling ==> Goal: opened(brian)	
Selection-by-purpose ==> Operation: load-slow(brian)	
Preconditions satisfied, so do it! Command: select-and-load(brian) display changes, so user's knowledge changes; user keeps track of which folder is opened:	**opened(brian)** **displayed-folder-name(brian)** **selected-to(brian)** **selected-from(brian)**
Preconditions on highlight-message satisfied, so do it! Command: pick(b1) display changes, so user's knowledge changes:	**highlighted(b1)**
Filtering precondition on move-quick no longer satisfied, so cease to be committed to it. Selection-by-purpose ==> Operation: move-slow(projectX,b1)	
Preconditions satisfied, so do it: Command: select-and-move(projectX)	**contains(projectX,b1)** **displayed-folder-name(projectX)** **selected-to(projectX)** **selected-from(projectX)**
...and so on to completion of task.	

Doing an IL analysis can often involve investigating several variants — establishing the consequences of different user knowledge on the predicted interactive behaviour. We move on to consider a variant analysis in which the user relies on predicting the information, because she is familiar with the operation of the device. (Here, the argument is that while novice users refer to the display frequently to confirm the effects of their actions, more experienced users often rely more on their ability to predict the effects of actions without explicitly checking them.) We can generate an IL description in which we do not declare that the user knows the selected-to folder or the selected-from folder by looking; rather, both are predicted effects. Most of the information in the second variant is the same as above. The important changes to the IL description are highlighted.

```
RELATIONS
highlighted(message)    -- detected-by-looking.
opened(folder)
selected-to(folder)
      -- selected-to(F) if displayed-folder-name(F)
selected-from(folder)
      -- selected-from(F) if displayed-folder-name(F)
visible(message)              -- detected-by-looking
contains(folder,message)      -- contains(F,M) if opened(F) & visible(M)

OPERATIONS
operation move-quick (folder:Tf, message:Ms)
      user-purpose:           contains(Tf,Ms)
      subgoaling-precond:     highlighted(Ms)
      filtering-precond:      selected-to(Tf)
      predicted-effect:       contains(Tf,Ms)
      action:                 click-move

operation move-slow (folder:Tf, message:Ms)
      user-purpose:           contains(Tf,Ms)
      subgoaling-precond:     highlighted(Ms)
      filtering-precond:      not selected-to(Tf)
      predicted-effect:       selected-to(Tf) &
                              contains(Tf,Ms)
      action:                 select-and-move(Tf)

operation load-quick (folder:Ff)
      user-purpose:           opened(Ff)
      filtering-precond:      selected-from(Ff)
      action:                 click-load

operation load-slow (folder:Ff)
      user-purpose:           opened(Ff)
      filtering-precond:      not selected-from(Ff)
      predicted-effect:       selected-from(Ff) &
                              opened(Ff)
      action:                 select-and-load(Ff)
operation highlight-message (folder:Ff, message:Ms)
      user-purpose:           highlighted(Ms)
      subgoaling-precond:     opened(Ff)
      filtering-precond:      contains(Ff,Ms)
      action:                 pick(Ms)
```

The corresponding modelled behaviour is shown in the table below.

Committed operations & goals, and commands issued	User's knowledge of state
...we get to a point where.... Goals: contains(projectX,b1)... contains(projectX,b6)	contains(projectX,a1)... contains(projectX,a5) contains(brian,b1)...contains(brian,b6) opened(amanda) highlighted(some-message) selected-to(projectX) selected-from(amanda)
Selection-by-purpose ==> Operation: move-quick (projectX,b1)	
Precondition-subgoaling ==> Goal: highlighted(b1)	
Selection-by-purpose ==> Op'n: highlight-message (brian,b1)	
Precondition-subgoaling ==> Goal: opened(brian)	
Selection-by-purpose ==> Operation: load-slow(brian)	
Preconditions satisfied, so do it! Command: select-and-load(brian) display changes, so user's knowledge changes; user tracks which folder is opened and the selected-from folder:	**opened(brian)** **selected-from(brian)**
Preconditions on highlight-message satisfied, so do it! command: pick(b1) display changes, so user's knowledge changes:	**highlighted(b1)**
Preconditions on move-quick satisfied, so do it! command: click-move	**contains(brian,b1) -- detected by looking** **contains(projectX,b1) --by tracking** *** see note (d) below **

According to this analysis, the experienced user is likely to:

a) do a slow-move to specify folder projectX as the "to" folder for the first message, then
b) a sequence of quick-moves to move further messages from folder amanda to folder projectX.
c) she will then do a slow-load to specify folder brian as the "from" folder, to open folder brian, and
d) resume doing quick-moves to move further messages from brian to projectX. The effect of this will in fact be to move files from brian to brian. Whether the user believes the message has moved will depend on whether she is relying on looking or predicting at this point. If she uses both, then a conflict will be detected.

The obvious solution is to have two separate *folder* slots on the display, one for "load" and one for "move". When this example was first used in training, one of the students described the use of one display slot for two purposes as "bad computer science karma"; the analysis here helps to give a user-centred view on *why* this was a poor design decision.

As is so often the case in interface design, this source of difficulty, like that discussed in Chapter 8, is obvious once it has been pointed out. However, these system descriptions are both based on systems that have been implemented and introduced into the marketplace. While neither of these usability problems is critical, both are annoying and unnecessary.

11. Discussion

We have used a series of examples to introduce you to the PUMA approach to cognitive modelling and usability evaluation. The examples have shown how the IL can be used to help the analyst identify mis-matches between the likely conceptual model and the device model, both in terms of the knowledge the user has access to and in terms of the ways they are likely to use that knowledge when interacting with the device. We close this tutorial by reviewing what PUMA is useful for and discussing how you can take PUM analysis further.

By now, you should be fairly familiar with the IL constructs. The syntax is given in Appendix 3. The aim in writing an IL description is always to lay out the knowledge a user needs to interact effectively with the device in order to achieve their domain goals. The boundary between "domain" and "device" knowledge, and the way the one relates to the other, can sometimes be fuzzy, but writing out the IL and considering where the user knowledge is expected to come from can help to clarify what the user needs to know, and how their domain knowledge is used when working with this particular device.

In the examples in Section 2, we have aimed to illustrate that the use of the IL can scale up and be used to analyse the usability of a broad range of devices. While doing a full analysis can be both difficult and costly, one of the potential advantages of PUM is that it can be applied with differing degrees of rigour, depending on what stage the design is at, or how many usability difficulties are identified early on.

The worked examples were chosen to illustrate the use of the IL to identify both surface-feature difficulties and deeper ones. In both of these cases, the need to present a detailed problem description, to allow the reader to do the IL-based analysis, probably means that the difficulties were fairly obvious without bothering to produce the IL. That is not because the difficulties are inherently "obvious" (these are, after all, descriptions of widely available commercial products), but because the need to describe them for analysis has taken the reader a large part of the way to understanding the systems — an important part of the learning process.

The scope of PUMA was discussed at the end of section 1. Broadly speaking, it is useful for analysing goal-directed behaviour, and for identifying mis-matches between likely user knowledge (and the way the user is likely to apply that knowledge when working with the device) and the design of the device.

Like any short tutorial, this one is inevitably incomplete. Learning to do cognitive modelling takes time and practice. Our aim has been to focus on the basics, so that you can apply your understanding in a range of contexts. In practice, we rarely construct a full IL description of substantial tasks, but use our understanding of what it would involve to guide a less formal user-centred analysis of usability.

One important aspect of interactive behaviour that we haven't covered in this tutorial is the idea that the device can act as a resource that guides or constrains what the user can do. So, for example, when using a menu system the user doesn't have to know ahead of time what option to select as long as the choice is clear when they actually use the device. In this case, the important thing is that the user can *recognise the relevance* of particular menu options is relation to their goals. The basic question, though, remains the same: what is the user expected to do, and how will they get the knowledge they need to do it with this device?

Appendix 1: trace of running model (text editor, first task)

This is a trace of a running model for the text editor (generated by a Lisp program — hence all the brackets!). At each stage, it shows the state of the device, and the state of the user model. "knows–ds" is what the user knows about the device state (from looking at the display, or predicting the effects of actions). Goals are adopted and commitments made to operations. When the user model requires input from the analyst (when there is a choice that it does not have the knowledge to resolve), the interaction is shown in italics.

```
device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc) (cursorat locd)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
  (cursorat locd)))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

adopting goals

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc) (cursorat locd)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
  (cursorat locd))
 (goals (textat tx-1 locc) (not textat tx-3 locc) (textat tx-1
locb)
  (not textat tx-2 locb)))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

considering ways to address goals ((textat tx-1 locc) (not textat tx-3 locc)
                                   (textat tx-1 locb) (not textat tx-2 locb))
the choice is: ((pastetext (textat tx-1 locc))
                (cut-from-text (not textat tx-3 locc))
                (pastetext (textat tx-1 locb))
                (cut-from-text (not textat tx-2 locb)))
(pastetext (textat tx-1 locc)) y/n? n
(cut-from-text (not textat tx-3 locc)) y/n? n
(pastetext (textat tx-1 locb)) y/n? y

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc) (cursorat locd)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
  (cursorat locd))
 (committed (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

subgoaling
the choice is: ((locateat locb (cursorat locb)) (copy-to-buffer
(buffercontents tx-1)))
(locateat locb (cursorat locb)) y/n? n

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc) (cursorat locd)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
  (cursorat locd))
 (committed (copy-to-buffer (buffercontents tx-1))
            (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
subgoaling
the choice is: ((marktext tx-1 (markedtext tx-1 (textat tx-1 l))))

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc) (cursorat locd)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
  (cursorat locd))
 (committed (marktext tx-1 (markedtext tx-1 (textat tx-1 l)))
  (copy-to-buffer (buffercontents tx-1)) (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

subgoaling
the choice is: ((locateat (textat tx-1 l) (cursorat (textat tx-1 l))))

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc) (cursorat locd)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
  (cursorat locd))
 (committed (locateat (textat tx-1 l) (cursorat (textat tx-1 l)))
  (marktext tx-1 (markedtext tx-1 (textat tx-1 l)))
  (copy-to-buffer (buffercontents tx-1)) (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (locateat (textat tx-1 l))  ( (loca) )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((cursorat loca) (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc)
 (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (cursorat loca) (textat tx-1 loca) (textat tx-2 locb)
  (textat tx-3 locc))
 (committed (marktext tx-1 (markedtext tx-1 loca))
  (copy-to-buffer (buffercontents tx-1)) (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (marktext tx-1)  ( (tx-1) )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((markedtext tx-1 loca) (cursorat loca) (textat tx-1 loca) (textat tx-2 locb)
 (textat tx-3 locc) (buffercontents tx-3))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (markedtext tx-1 loca) (cursorat loca) (textat tx-1 loca)
  (textat tx-2 locb) (textat tx-3 locc))
 (committed (copy-to-buffer (buffercontents tx-1)) (pastetext (textat tx-1
locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (copy-to-buffer)  ( nil )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((buffercontents tx-1) (markedtext tx-1 loca) (cursorat loca) (textat tx-1
loca)
 (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (buffercontents tx-1) (markedtext tx-1 loca) (cursorat loca)
  (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
```

```
 (committed (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

subgoaling
the choice is: ((locateat locb (cursorat locb)))

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((buffercontents tx-1) (markedtext tx-1 loca) (cursorat loca) (textat tx-1
loca)
 (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (buffercontents tx-1) (markedtext tx-1 loca) (cursorat loca)
  (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
 (committed (locateat locb (cursorat locb)) (pastetext (textat tx-1
locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (locateat locb)  ( (locb) )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((cursorat locb) (buffercontents tx-1) (textat tx-1 loca) (textat tx-2 locb)
 (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (cursorat locb) (buffercontents tx-1) (textat tx-1 loca)
  (textat tx-2 locb) (textat tx-3 locc))
 (committed (pastetext (textat tx-1 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (pastetext)  ( nil )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 locb) (cursorat locb) (buffercontents tx-1) (textat tx-1 loca)
 (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
 (committed))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

adopting goals

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 locb) (cursorat locb) (buffercontents tx-1) (textat tx-1 loca)
 (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
 (committed)
 (goals (textat tx-1 locc) (not textat tx-3 locc) (not textat tx-
2 locb)))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

considering ways to address goals ((textat tx-1 locc) (not textat tx-3 locc)
                                   (not textat tx-2 locb))
the choice is: ((pastetext (textat tx-1 locc))
                (cut-from-text (not textat tx-3 locc))
                (cut-from-text (not textat tx-2 locb)))
(pastetext (textat tx-1 locc)) y/n? n
(cut-from-text (not textat tx-3 locc)) y/n? n


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
((textat tx-1 locb) (cursorat locb) (buffercontents tx-1) (textat tx-1 loca)
 (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
 (committed (cut-from-text (not textat tx-2 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

subgoaling
the choice is: ((marktext tx-2 (markedtext tx-2 locb)))

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-1 locb) (cursorat locb) (buffercontents tx-1) (textat tx-1 loca)
 (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
 (committed (marktext tx-2 (markedtext tx-2 locb))
  (cut-from-text (not textat tx-2 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (marktext tx-2)  ( (tx-2) )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((markedtext tx-2 locb) (textat tx-1 locb) (cursorat locb) (buffercontents tx-
1)
 (textat tx-1 loca) (textat tx-2 locb) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (markedtext tx-2 locb) (textat tx-1 locb) (cursorat locb)
  (buffercontents tx-1) (textat tx-1 loca) (textat tx-2 locb) (textat tx-3
locc))
 (committed (cut-from-text (not textat tx-2 locb))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (cut-from-text)  ( nil )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((buffercontents tx-2) (textat tx-1 locb) (cursorat locb) (textat tx-1
loca)
 (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-3 locc))
 (committed))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

adopting goals

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((buffercontents tx-2) (textat tx-1 locb) (cursorat locb) (textat tx-1 loca)
 (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-3 locc))
 (committed) (goals (textat tx-1 locc) (not textat tx-3 locc)))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

considering ways to address goals ((textat tx-1 locc) (not textat tx-3 locc))
the choice is: ((pastetext (textat tx-1 locc))
                (cut-from-text (not textat tx-3 locc)))
(pastetext (textat tx-1 locc)) y/n? y
```

```
device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((buffercontents tx-2) (textat tx-1 locb) (cursorat locb) (textat tx-1 loca)
 (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-3 locc))
 (committed (pastetext (textat tx-1 locc))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

subgoaling
the choice is: ((locateat locc (cursorat locc)))

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((buffercontents tx-2) (textat tx-1 locb) (cursorat locb) (textat tx-1 loca)
 (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-1 locb) (cursorat locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-3 locc))
 (committed (locateat locc (cursorat locc)) (pastetext (textat tx-1
locc))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (locateat locc)  ( (locc) )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((cursorat locc) (buffercontents tx-2) (textat tx-1 locb) (textat tx-1 loca)
 (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (cursorat locc) (textat tx-1 locb) (buffercontents tx-1)
  (textat tx-1 loca) (textat tx-3 locc))
 (committed (pastetext (textat tx-1 locc))))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

preconditions satisfied, so doing it!
issued command (pastetext)  ( nil )


device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-2 locc) (cursorat locc) (buffercontents tx-2) (textat tx-1 locb)
 (textat tx-1 loca) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-2 locc) (cursorat locc) (textat tx-1 locb)
  (buffercontents tx-1) (textat tx-1 loca) (textat tx-3 locc))
 (committed))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

adopting goals

device-state: -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
((textat tx-2 locc) (cursorat locc) (buffercontents tx-2) (textat tx-1 locb)
 (textat tx-1 loca) (textat tx-3 locc))
user-state: - - - - - - - - - - - - - - - - - - - - - - - - -
((knows-ds (textat tx-2 locc) (cursorat locc) (textat tx-1 locb)
  (buffercontents tx-1) (textat tx-1 loca) (textat tx-3 locc))
 (committed) (goals (textat tx-1 locc) (not textat tx-3 locc)))
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

considering ways to address goals ((textat tx-1 locc) (not textat
tx-3 locc))
the choice is: ((pastetext (textat tx-1 locc))
                (cut-from-text (not textat tx-3 locc)))
```

Appendix 2: Heuristics for the IL

GENERAL

G1. Writing the IL is an iterative activity. Don't even try to get it right first time. Start with whatever part seems easy: objects, operations, device, whatever.

G2. Choose as simple a representation as you can, that captures the important aspects of the design you are analysing.

G3. Don't be too clever! Include only what the user can reasonably be expected to know, and not any complex mathematical relationships.

G4. Keep a good balance between formality and informality. You can typically afford to be more casual on the device side than on the user side, provided that the behaviour of the device is clear. But give enough detail on the device to connect up with the user side, e.g. at least list the device commands and the information displayed.

G5. If there's something important (especially on the user side) but you don't know how to express it in the IL, then write it anyway, e.g. as a comment.

G6. Remember that if there's something you find difficult to express in the IL, then that aspect may be hard for the user to cope with too.

SYNTAX and CLOSURE

C1. Check that each variable used in an operation is defined ("bound") somewhere, e.g. in the arguments to the operation, or in the user-purpose.

C2. Check that for each relation in a subgoaling precondition, there is an operation to achieve it.

C3. Check that for everything the user needs to know, there is a way for it to be known (i.e. either visible, or through being a predicted effect of some operation).

C4. Use subgoaling preconditions for relations the modelled user would try to make true; use filtering preconditions for ones that the user cannot, or would not want to, make true.

SPECIFIC IDEAS

S1. For generating operations, focus on the device commands.

S2. For generating operations, focus on the user's conceptual actions.

S3. Remember that conditions (i.e. subgoaling preconditions and filtering preconditions) are knowledge conditions, i.e. they refer to information known to the user. They may not necessarily correspond to the actual device state.

S4. The same information on the display can serve as the source for two or more relations.

S5. For information that is not visible and therefore needs to be predicted, assume that the user tracks only the main effect of an operation — which will often coincide with its user-purpose — and not any side effects.

S6. If there is some information visible on the screen which is (a) frequently used, and (b) not particularly visually salient, then consider doing an alternative analysis in which the effect is assumed to be predicted instead of checked from the screen. This is particularly pertinent if the information is used for different purposes at different times.

S7. Remember that the same device command can correspond to different conceptual operations in different circumstances.

S8. Remember that the same user-purpose may be achieved by different device commands (and therefore different conceptual operations) in different circumstances. These different conceptual operations will generally (though not always) have different subgoaling preconditions or filtering preconditions.

Appendix 3: Summary of Instruction Language

========== *DECLARATIONS* ==========

OBJECTS

```
object-type: named-objects
```

RELATIONS

```
relation-name(object-type, object-type, ...)
                                          (detected by looking?)
```

========== *USER KNOWLEDGE* ==========

OPERATIONS

```
operation op-name (object-type:Variable, ...)
      user-purpose:           relations
      subgoaling-precond:     relations
      filtering-precond:      relations
      predicted-effect:       relations
      action:                 device command
```

USER KNOWS INITIALLY

```
instances of relations
```

DESIRED-STATE

```
instances of relations
```

========== *DEVICE DESCRIPTION* ==========

DEVICE COMMANDS

```
describe each command
```

INITIAL DEVICE STATE

```
instances of relations
```

DISPLAYED

```
Describe what information is displayed to the user
```