University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

Intelligent User Interfaces: Survey and Research Directions

Edward Ross

March 2000 | CSTR-00-004

Intelligent User Interfaces : Survey and Research Directions

Edward Ross *Edward.Ross@bristol.ac.uk*
Department of Computer Science, University of Bristol
Merchant Venturers Building, Woodland Road
Bristol, BS8 1UB, United Kingdom

January 2000

Abstract

Computers are extremely effective information processors. However, for current, highly interactive usage, computers can only be as effective as the interface which is used to communicate with them. This report first examines the development of the user interface to place the intelligent interface in historical context. Following this is a survey of intelligent interface paradigms and techniques. These show how new interfaces can improve communication between humans and machines when the interface technology makes the leap from a passive tool-set to a pro-active assistant. These assistants, or agents, are viewed in the context of a new type of interactive, partial solution to previously insoluble problems.

1 Introduction

This report examines intelligent user interface technology, including their utility and implementation methods. Section 2 looks at the development of the human machine communication, from command driven shells and direct manipulation graphical user interfaces, through to intelligent interfaces. An overview of intelligent user interfaces is given in Section 3. If an intelligent user interface maintains some kind of model of the user then this user model can be used to automatically adapt the interface. Thus user models are an important component in many intelligent interfaces and Section 4 is devoted to user modelling. Following this is a survey of intelligent interfaces in Section 5. Concluding remarks about intelligent user interfaces can be found in Section 6.

2 Human Machine Communication

The basic operations of computers are numerical calculations. As a mathematical tool computers are highly effective, quickly handling large quantities of data

without making any calculation errors. Furthermore, computers are powerful information processing tools. In order for a human to use a computer to process information it is necessary to communicate both the processing task and the input data to the computer. It is essential for this communication to be both effective and efficient since it underpins the entire operation of the computer as an information processing tool.

Any problem solvable by the computer must be expressible using machine code, but this is notoriously difficult for humans to do. Higher level languages and libraries of standard functions provide more effective communication than machine code. However, the majority of users do not have the skills to program a computer, furthermore programming is always a time consuming task. Obviously, programs only need to be written once, but to use existing programs, users must still communicate input data and specify which program should be used. For this a user interface is required.

At the dawn of computing programs and input data were communicated on punched cards and the output would be printed. Later the keyboard was used for typed input, and the cathode ray tube (CRT) monitor was used for visual output. A shell was the first type of user interface, being used in conjunction with these devices. A shell is a textual interface which allows users to enter commands to start programs and specify data values. Early computers were used primarily for scientific or mathematical purposes, which only requires low interactivity. The human operator need only specify input data and then wait for the computer to produce the output data. A shell is an adequate interface for these purposes.

As time went on programs became more general, designed to solve a wider range of tasks. Application programs emerged such as word processors (for producing formatted text) and spreadsheets (for handling general mathematical calculations). These application programs necessarily involve a high level of interactivity between human and machine. Unfortunately communicating with applications through a textual interface is somewhat cumbersome. The Graphical User Interface (GUI) provided a much more effective and natural interface between humans and their computers, aided by an additional input device, the mouse. The GUI uses images of objects as metaphors for real world objects; these objects are often called icons and they exist within a metaphor for the desktop.

If one examines the way in which actions and objects are specified using the interface, it can be seen that the style of interaction in a GUI is different from a shell. In a shell the user enters an action and then the objects to which that action will be applied. In a GUI the user selects the objects, and then selects the action to perform. This ordering of object and action selection means that a GUI allows users to directly manipulate desktop objects using the mouse to point at, click on and drag icons (this contrasts with shell usage, where the user issues commands which have a subsequent effect on the interal state of the computer). This use of directly manipulatable graphical objects within the desktop makes it easier for users to understand the effects of their actions, and consequently easier to plan how to achieve their goals.

Current implementations of graphical user interfaces certainly increase the effectiveness of human machine communication and leverage interactivity, but are not without limitations. The primary drawback is that in general, application interfaces are fixed and rigid; in other words software is designed in a 'one-size-fits-all' manner [19]. Many applications do provide preferences which allow users to customise the interface to some extent. However, the interface is still entirely passive. It is left to the user to set the preferences. This assumes that users realise that the preferences exist, that they understand them and have the time and inclination to alter them. These problems can be alleviated by an intelligent interface, which provide a more active style of interface.

3 Overview of Intelligent User Interfaces

Intelligent interfaces can be divided into three classes:

- Adaptation within direct manipulation interfaces
- Intermediary interfaces
- Agent interfaces

The first of these involves the addition of adaptation to an existing direct manipulation interface. This can be achieved by adding extra interface objects, intended to hold the predicted future commands. Alternatively, one can design an interface where multiple commands, which are not likely to be used, are folded up into single objects. To take adaptation further it is necessary to change the nature of the interaction. In order for the interface to filter information[1] (informative interface) or generate suggested data values (generative interface), the interface must act as an intermediary between the user and the direct manipulation interface.

It would be useful if intelligent interfaces could make suggestions, correct misconceptions, and generally guide their users [15]. To achieve this it is sensible to make the intermediary explicit as an autonomous agent. The user retains full control over the directly manipulatable interface and can see the agent, so should not be confused by the fact that the agent acts autonomously. This style of interaction has been referred to as indirect-management [1].

Autonomous agents may serve a variety of purposes, but a key feature is that they provide pro-active support to the user. The agent will typically assist by making suggestions, since this is non-invasive. In some cases the user can cause the agent to carry out its suggested actions; this is precisely the way in which programming by demonstration systems operate. Some agent systems, such as help systems, exist purely to make suggestions and give advice.

[1] It is worth noting the increasing use of multimedia in computing, made possible by GUIs. Accessing multimedia, in particular through the world wide web has revolutionised computer usage in the 1990s. This vast collection of easily accessible data drives the need for information filtering systems. Adaptive user interfaces can act as effective personal information filters.

The interface as agent paradigm is part of an emerging set of solutions[2] in computing which aim to assist their human users, rather than replace them. This approach allows computers to be of use in areas that are insoluble using traditional methods. The underlying concept behind computer assistance, is that the computer can learn the repetitive or mundane parts of the task (which can be solved using computational techniques), leaving human users to do the parts which require thinking that is less obviously logical. Currently this kind of thinking is not fully understood, it could be referred to as intuitive thinking or subconscious thinking. Moreover, it is not known for certain whether human thought is purely analytical, some argue that there are non-analytical sub-conscious modes of thought [7].

3.1 Adaptive User Interfaces

Many intelligent interfaces can be viewed as adaptive interfaces, since they change their behaviour to adapt to a person or task. Adaptive user interfaces are set to play a major role in the next generation of computer interfaces. This is primarily due to the high level of interactivity in modern software. Whilst there have been significant developments in human computer interaction (HCI) over time, software still tends to have a fixed interface, which behaves in the same way, regardless of the individual user. These rigid, inflexible interfaces do not reflect the differences inherent in their human users. People have diverse working methods, levels of experience and expertise and this clearly points to a need for the software interface to be adaptable [13].

Adaptive user interfaces use machine learning to improve their interaction with humans. The learning is directed through user models, these are discussed in Section 4. This enables the formation of an interface tailored to the abilities, disabilities, needs and preferences of the individual user.

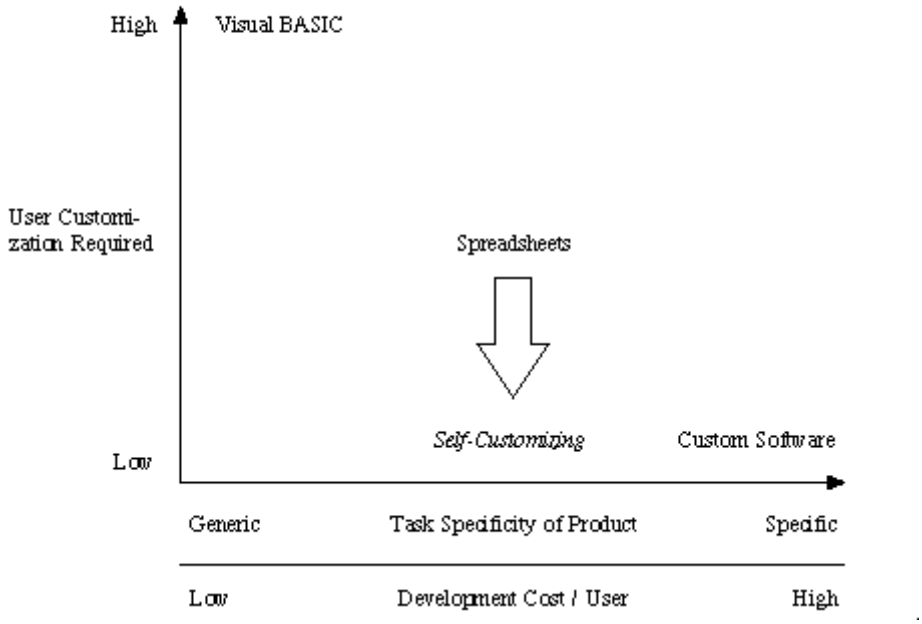
Adaptation should be designed to aid the user to reach their goal more quickly, more easily or to a higher level of satisfaction. The adaptive system may satisfy one, or all of these requirements, but must not infringe on the others. Furthermore, the system must not irritate the user in any way. From this we see that the adaptive system must not slow the computer down or perform any unwanted operations since this will slow down progress towards the user's goal and cause irritation.

3.2 Cost Benefits of Adaptive User Interfaces

Implementing an adaptive user interface can reduce application development cost per user. The graph [2] in Figure 1 plots user customisation required on the vertical axis against development cost per user and task specificity of product on the horizontal axis. The more general the application, the higher the user

[2] An example of a solution which aims to assist the user (rather than replace the user) would be amorphous program slicing for debugging, which transforms programs into more effective form for the user to perform the debugging.

Figure 1: User customisation required against task specificity of product and development cost per User



customisation required. However more general products have a larger user base, so the development cost per user is lower for more general applications.

At one end of the spectrum we have programming languages, such as Visual Basic. These can be applied to any task, but a large amount of user customisation is required (here customisation is actually programming). At the other end of the spectrum sits custom software, which obviously requires no user customisation and is task specific. In between lie applications such as spreadsheets (which can be applied to a range of mathematical tasks). Clearly self-customising software does not require user customisation, and it can be moderately generic before customisation. This means that self-customising software delivers the benefits of custom software at a lower development cost per user.

4 User Modelling

The user model is an integral part of any intelligent user interface aiming to adapt to suit its users. A *user model* is an explicit representation of the properties of an individual user; it can be used to reason about the needs, preferences or future behaviour of the user. Most computer systems that interact with humans

contain some kind of implicit model of the user. For example, a program using the mini-max game playing strategy has an implicit model of a single canonical user; it assumes that the user will always make the best possible move. This model affects the program, but is not stored explicitly. The paper 'User modelling via stereotypes' [18] is widely regarded as marking the beginning of user modelling, Rich investigates the possibilities for allowing computers to treat their users as individuals with distinct personalities and goals.

User modelling is now a mature field and demonstrably useful research systems exist. Yet few commercial implementations of user modelling exist since there are some difficulties encountered when incorporating a user model into the design of existing systems. User models are sometimes quite complicated, meaning implementation requires significant development resources. This problem is exacerbated by a lack of empirical studies of user model performance benefits and effects on user training. Performance overheads must also be taken into account. These problems have motivated research into minimalist user models [20], which aim to provide the benefits of user modelling with a very simple modelling system, cutting down on development costs and performance overheads.

4.1 User Model Design

There can be a wide variety of user model types, and models can be classified along the four major dimensions listed below.

- What is modelled: Canonical user - or - Individual user
- Source of modelling information: Model constructed explicitly by the user - or - Abstracted by the system on the basis of the user's behaviour
- The time sensitivity of the model: Short-term, highly specific information - or - Longer-term, more general information
- Update methods: Static model - or - Dynamic model

The update methods often follow from the other three dimensions; individual user models, models abstracted on the basis of user behaviour and short-term models generally require dynamic update. If the model contains very short-term information then it can become a task model, since it is relevant to the task in hand, and the individual user is not important. This is because the model will update immediately to reflect any task which a new user undertakes.

The most basic type of model is static and contains a canonical user. This type of model can be embedded within a system and almost does not need to be stored explicitly. In contrast, if the individual user is modelled then dynamic update is required, and explicit methods are necessary to describe how the user model state affects the system performance.

The core of a user modelling system is the User Synopsis (USS), the USS can gain modelling information from two sources; the modelling information can

be elicited directly from the user by posing questions, or the system can watch the user's interactions and abstract the model from these. Watching the user's interaction with the computer is sometimes referred to as 'Observing the user through the keyhole of the interface' [10, 6]. Using direct questioning is generally regarded as a less desirable solution because it places additional demands on the user's time. However, it is suitable in systems which can operate primarily using information gained from watching the user, relying on direct questioning only for additional information.

4.2 Learning the USS

The data actually stored in the USS can be any attribute of an individual which is potentially relevant to the task in hand. The values which these attributes take are usually learned. To illustrate this learning some example systems are now examined.

GRUNDY is a book recommendation system [18]. In Grundy the USS stores personal attributes related to the books a person enjoys reading. Example attributes are political standpoint, level of education and preferred plot speed. It is not obvious how such USS attributes are linked to observed interface behaviour, so it is expedient to examine this in greater detail.

GRUNDY uses stereotypes[3] to update the USS. Each stereotype is a collection of attributes which a group of people may share in common; example stereotypes are man, woman, scientist, artist etc.. The stereotypes are arranged in a Directed Acyclic Graph (DAG), with a general to specific ordering, the most general stereotype being any-person, which has a value for every characteristic. Each stereotype inherits all the values from stereotypes more general than itself, with more specific values overriding more general values. Each stereotype can have various triggers; a trigger links an observable event to a stereotype. A trigger also holds a weight, which determines how strongly the stereotype affects the existing USS. So for example if it is observed that a person uses a complex operation in an application, then a trigger for the expert-user stereotype could be fired with high confidence. If the human uses technical terms in a natural language interface then the technical-person stereotype could be activated.

Under the generally accepted definition of learning the system must make generalisations based on partial experience in order for it to be considered to 'learn'. It is not sufficient to simply remember past interactions [13]. According to this definition of learning, stereotypes are not a machine learning approach to user modelling. It does not suffice that stereotypes are generalisations, since but they are not generated during use of the user model. User modelling systems using stereotypes are learned systems rather than learning systems. However, it is possible to combine stereotypes with learning systems. Stereotypes form good initialisation values for the USS, and once initialised the USS can change with experience, using machine learning techniques to form new generalisations.

Davison et al employ user modelling to predict the next command that will

[3] For fuller details on the usage of stereotypes see [18].

be issued at a Unix command prompt [5]; here the USS stores a probability for each possible command. These probability tables are not explicitly referred to as user models, but the performance is specific to the individual user, so the probabilities are, in fact, personal attributes. In the case of command prediction, abstracting the model is very straightforward. The personal attributes are probabilities that the various commands will be issued. Upon observing any particular command, the probability of observing that command again is increased. All command probabilities are then decayed to give higher probabilities for more recent commands.

There will always be an element of uncertainty in the user model, thus some form of reasoning under uncertainty is required. Probabilistic techniques, in particular Bayesian networks, have proved effective for user modelling. The Lumière project explored the use of 'Bayesian user modelling for inferring the goals and needs of software users' [10]. The project resulted in a language for describing high level network events in terms of low level user events and also produced the prototype for the Microsoft Office Assistant[4], an intelligent help agent.

5 Survey of Intelligent User Interfaces

This section provides some examples of intelligent user interfaces. The samples are not exhaustive, but instead provide a selection of interesting systems. The interfaces are split into four classes, direct manipulation interface adaptation, informative interfaces, generative interfaces and programming by demonstration. The first class, direct manipulation interface adaptation, is obviously the closest to existing systems, this is my own terminology and probably doesn't have a great deal of relevant literature. The informative and generative interface classes are described by Langley in two papers on Adaptive User Interfaces [12, 13], these classes must either be intermediaries or autonomous agents. The fourth class, Programming by Demonstration (PBD), is a research area in its own right, a good reference is Cypher's book 'Watch What I Do' [4].

Adaptation with direct manipulation and informative interfaces improve interaction by reducing the branching factor of solutions. This can be achieved by focusing the user's attention on a subset of the available commands, or by filtering information before it is presented to the user. Generative interfaces or programming by demonstration are distinct because they aim to reduce the length of solutions by inducing data values based on previously observed values.

[4]Many users find the Microsoft Office Assistant irritating, this detracts from their appreciation of its powerful intelligent help system. It is important to note that key user model features were not included in the Office Assistant. Critically the part of the model that predicted the likelihood that the user would mind being interrupted was not included!

5.1 Direct Manipulation Interface Adaptation

The user model can be used to predict the user aims and preferences or spot behavioural patterns. Armed with these predictions the next generation of software could adapt in four main ways, as listed below.

- Speculative Execution - The predicted actions can be carried out speculatively, such that when the user issues a command, the necessary steps are already under way or even completed.
- Pattern Completion - If clear behavioural patterns are evident then several commands can be combined into a single meta-command.
- Rapid Issue - The commands that execute the predicted actions or action sequences can be made easily accessible to the user for fast issue.
- Assistance - The system can offer help and assistance based on its knowledge of the user's aims.

Basic forms of adaptation already exist in some widely used commercial software. For example the MS Windows 95 operating system and MS Office 97 productivity suite provide recently used file menus. Coupled with the fact that the user can have a separate profile under Windows, the recently used file menus can be viewed as a form of personalised adaptation. However, since the menus just provide the n-most recently used files, there are no generalisations and no induction used. A significant level of adaption is provided by MS Office 2000, in all the pull down menus, less frequently used commands are wrapped up into special expandable menu entries based on frequency of use.

More sophisticated command prediction systems have been created as research systems. Davison and Hirsch created a system for predicting Unix command stubs (the command with no arguments) [5]. This system is based on a probabilistic user model, containing likelihoods for each command given the previously issued command. The system displays the top five predictions, which can be selected using the function keys. In tests, one of the five predictions is correct 75% of the time. Clearly when selecting the number of commands to list, there is a trade off between cognitive load for the human user and likelihood of listing the correct command.

Davison and Hirsh's system uses their Incremental Probabilistic Action Modelling (IPAM) algorithm. This is similar to a naive Bayes classifier but decays probabilities with time. The decay gives emphasis to recently issued commands and allows the algorithm to handle the concept drift inherent in command prediction.

Korvemaker and Greiner [19] extended this system to include full Unix commands and take into account temporal information and error codes. A method is required to handle the complexity of this task which was estimated to require a table with approximately three billion entries, even with error codes stored with two values and time quantised to hours. Consequently, only a subset of

'current' commands was maintained and the temporal information was quantised into morning, afternoon, evening and night. In tests, this system obtained nearly 50% predictive accuracy over the full Unix command.

5.2 Informative Interfaces

The amount of information currently accessible via computers is huge, principally because of the Internet and the world wide web. In light of this vast quantity of available data, the need for information filtering is clear. There are two main methods for filtering information, content-based, and collaborative filtering [12].

Content based filtering represents each object with a set of descriptors, such as words in a document. By observing which objects the user accepts or rejects, the system can learn which descriptors indicate a likely positive or negative user interest in an object. An example of content based system is SYSKILL & WEBERT [17]. Syskill & Webert recommends web pages, on a given topic, that the user is likely to find interesting.

Collaborative information filtering (also referred to a social information filtering) can be used for objects where descriptors cannot be readily identified, or where user interest can only be measured subjectively. A collaborative filtering system must first identify a set of objects in which the user takes a positive or negative interest. Then the system needs to find another user profile where likes and dislikes intersect. Further objects of potential interest can then be suggested from the intersecting profile. Using this method, subjective media such as art work, music or films can be filtered. Collaborative filtering is used in FILMFINDER (available at URL:`www.filmfinder.com`), which recommends films that users might like.

Content based and collaborative filtering are not mutually exclusive. A combination of these methods is used in WISEWIRE [5]. This is used to recommend news stories and web pages. The interface derives from Lang's NEWSWEEDER [11]. Content based filtering learns to predict topics which the user has previously taken an interest in, while the collaborative filtering can predict topics which the user hasn't yet seen but may be of interest.

5.3 Generative Interfaces

A generative interface creates new data on the basis of previously observed data values. By inducing data values, the system can reduce data entry time for a human operator and potentially improve the quality of the data. This is similar in function to learning apprentices, which observe an expert in order to improve the performance of a learned computer system for a particular task. The emphasis of a generative interface is on using learning to improve a human's performance at a task.

[5] WISEWIRE was purchased by Lycos on April 22, 1998. WiseWire technology has now been integrated into Lycos search technology, available from URL:`www.lycos.com`.

An example of a generative interface which aims to improve the quality of data entry is CLAVIER [9]. Clavier is a system for assisting in the process of curing parts in an autoclave. It deals with loads and layouts for aircraft parts. When presented with a set of parts the system retrieves examples from a case library, selecting an arrangement which has cured well and which contains many of the required components. An example of a system aiming to improve speed of data entry is Hermens and Schlimmer's generative interface for filling out repetitive forms [8]. The system suggests default values for fields in the form, which the user can override. Each new form the user completes is used as a training example for the system.

Generative interfaces are similar to programming by demonstration systems (described in Section 5.4). Generative interfaces produce data values whereas programming by demonstration systems generate commands with arguments. Both types of system tend to focus on constrained tasks, learning on-line from just a few training examples.

5.4 Programming by Demonstration

Programming By Demonstration (PBD) [4] is a method of communicating simple iterative programs to a computer, without actually needing to write the program. The user can demonstrate the required actions and the PBD system induces the rest of the operations. The most basic programming by demonstration system is a macro recorder. The macro recorder cannot however make generalisation from one iteration to the next.

Below is a list of considerations when building a programming by demonstration system:

- Representation of User Actions
- Representation of Predictions
- Loop Detection
- Generalisation
- Domain Knowledge
- Validating Programs
- Termination Conditions
- Level of Intrusion

The first item, representation of user actions concerns the input data to the PBD system. The data should be of a suitably high level, so that the system can make meaningful predictions. For example button or menu item clicks provide far superior data than positions of mouse clicks.

An important PBD design decision is how to represent the systems predictions to the user. These could be presented in the same format as the system sees

the events, as textual descriptions, for example something of the form 'Mouse click on button ID 34, process ID 83'. The clear disadvantage with text being that the user needs to be able to interpret the predictions. It is better to use the same modality that the user employs for performing actions. This can be achieved by highlighting the GUI objects which are going to be 'clicked on' by the prediction scheme.

The loop-detection and generalisation are related to the machine learning aspect of PBD. It is possible to perform loop detection and generalisation between loops as two separate phases. Alternatively, first-order learning could be used, in which case both would be part of the same learning scheme. It is generally recognised that the level of domain knowledge has a high impact on the effectiveness of machine learning. To be useful, a PBD system will need to know something about the objects and object hierarchy in the application domain. This will allow the system to compare objects and identify iteration over groups of objects. However, it is also useful if the PBD system can function in the absence of domain knowledge. In fact one of the features of Davison and Hirsh's Ideal Online Learning Algorithm (IOLA) [5] is, '[that the algorithm should] apply even in the absence of domain knowledge'.

The domain knowledge also affects termination conditions. The termination of the loop is a pertinent problem in PBD. It is difficult or sometimes impossible to predict when to stop iterating using the first few iterations of the loop as examples. If the system has domain knowledge then it can use the structure of the data to determine halting conditions (for example once all the files in a directory had been processed the program could terminate).

The final criterion is the level of intrusion. That is, how much the PBD system intrudes on the user's normal interface usage. A minimally intrusive PBD system would never ask for information, such as requiring the user to signal the start or end of an example, or explicitly confirm the correctness of predictions.

Cypher's EAGER [3] is a system which solves iterative PBD. The system can be applied to general programs, but modification of the target application is required. This includes communication of events and data structures. Eager is designed to be minimally intrusive, never asking the user questions. It is always running so it can identify repetitions in user actions. When it identifies a pattern the Eager icon is displayed. Then, as the user continues to perform the task, Eager indicates its predictions by turning desktop objects green, including menu items, buttons, and text selections. The user can carry on with their task as per usual, so if Eager is wrong then the user will not loose any time. In the case that Eager makes an incorrect prediction then it will revise its model of the interaction. Once the user is satisfied that Eager is predicting correctly then he or she can click on the Eager icon to cause Eager to complete the task automatically.

A study of Eager was carried out on seven first time users. The study showed that first-time users were generally able to understand what Eager was doing and to work out how to use it without instruction. Eager could detect patterns in user actions even when the users chose different strategies for performing

tasks, or when the subjects performed the task somewhat differently on each iteration.

The most striking negative finding was that all subjects were not entirely comfortable about giving up control when Eager took over. A further problem was that some subjects did not notice the Eager icon when it first appeared (indicating that a pattern has been discovered), particularly when displayed on a rich background. To draw the user's attention the Eager icon was animated when it first displayed. Overall the study showed that PBD can be applied practically to current programs.

Paynter and Witten's FAMILIAR [16] performs a similar function to EAGER. Familiar extends the state of the art because it can operate with existing application interfaces. It does not require modifications to the application, nor does it need a model of the application. Familiar can also learn tasks which span applications. It can operate with multiple applications because it uses the standard APPLESCRIPT for communication.

This kind of manipulation, by emulating the user, has been described by Lieberman [14] as a "marionette strings" approach. It is his view that this approach works but would be much improved by writing applications that support closer integration of data between the application and agent.

6 Summary and Open Issues

There are three core issues in user interface design: ease of use, power, robustness. When compared with the shell interface, it is generally acknowledged that the Graphical User Interface (GUI) improved ease of use; which for the majority of computer users is very important. However, in some cases the GUI reduces the power of the interface. To give one example, it is hard to move all files with a particular character string within their name from one directory to another using a GUI interface, but this is easy using a shell. Moreover shells are often more robust than GUIs; this is to be expected since a shell is less complex than a GUI and hence likely to contain less bugs.

The new interface paradigm of autonomous agents is driven by the need to improve ease of use for an ever broadening, often untrained, user base. Interestingly, agents have the potential to improve ease of use while simultaneously increasing the power of the interface. As an untiring apprentice, the agent can learn to perform mundane, repetitive tasks. This is referred to as Programming by Demonstration (PBD). Ironically, despite the fact that computers are 'good' at performing straightforward, repetitive tasks, they can often provide no alternative to performing the task manually, using the existing interface tools. This is because the effort of communicating the details of a task to a computer using conventional programming, is usually great, even for experienced programmers. As a result of this, PBD is useful to both experienced and inexperienced user alike. In other words PBD is not just a tool for beginners who don't know how to program computer systems.

The most fundamental drawback of autonomous agents is their hunger for

computational resources, as required for difficult tasks such as learning and reasoning under uncertainty. However, there is no reason to suppose that an agent needs to share a processor with other programs. Separate processors and memory units can be used for separate functions. Indeed there are currently many commercially available multiprocessor computer systems. Even where individual applications have not be designed to use more than one processor, multiple processors can still be utilised. As in the human brain, parallelism can be introduced for separate tasks such as speech recognition, speech generation, autonomous agent reasoning systems, the Operating System (OS) and standard application programs.

A further problem with autonomous agents acting independently is the user's reluctance to give up control to the agent [3]. The user's trust will only be won by agents that function correctly. They must also demonstrate their capabilities in an unintrusive manner, until such a point that the user is confident in the capabilities of the agent. Since each user will evaluate 'correct' functioning of the agent under their own criteria it is likely that an agent with a user modelling component will be judged to have superior functionality over its non-adaptable counterpart.

PBD uses machine learning generalisation techniques for induction of a target program, given some examples of the target program steps. A user model, as described above, would act as an evaluation heuristic, against which induced target programs could be measured. Therefore a PBD system with such a modelling component could learn to learn programs which suited a particular user's working methods. To date research has not focused on this aspect of PBD. FAMILIAR [16] includes a learned model for selecting between prediction schemes. However, no online update mechanism is included in Familiar. An online learning system could react immediately, a strong requirement since such a fast reaction is required in the short term learning setting of PBD.

Learning to learn, or meta-learning, is an active area of current research. It focuses on combining and selecting results from multiple models. The learning in FAMILIAR is a form a meta-learning, in which the learned selection model represents a learning bias. Making this system incremental would allow the learning bias to change and therefore accommodate concept drift. Paynter compares his FAMILIAR system to the stacked generalisation ("stacking") method for combing machine learning models [21]. There is potential that recent meta-learning results could improve PBD system performance.

Probabilistic techniques are useful for reasoning under uncertainty, as is required for user modelling. Furthermore they lend themselves well to incremental implementations, because the probabilities can simply be updated in light of new example data. Bayesian networks may be a good candidate for a probabilistic learning method, because they provide a way of storing and reasoning about a probability distribution, which was previously computationally intractable without using the naive Bayes assumption.

Yet another challenge when implementing a PBD system is the discovery of termination conditions. Existing PBD systems are designed to take as input data the actions performed by the user in the first few iterations. In most cases

it is impossible to induce termination conditions from these input data. If one views the aim of PBD as providing a more effective mechanism for communicating a simple iterative program than the act of programming, then it can be seen that termination problems arise from the choice of input data. All that is required is that the user demonstrate the termination condition to the PBD system. This is possible whenever the input objects to the target program all exist prior to execution of the target program. This will frequently be the case in problems to which PBD systems are applicable, namely modifying a set of existing objects.

Existing PBD systems tend to use a two phase attribute learning approach. Phase one identifies interactions, and phase two generalises between iterations. An alternative approach would be to view PBD as a 1st order learning problem. To my knowledge such an approach has not yet been applied to programming by demonstration.

In conclusion, it can be seen that there exist many systems which demonstrate the potential of both adaptive user interfaces and programming by demonstration agents. Also, as stated above there are several interesting avenues of research open to investigation, within the area of programming by demonstration. These include the application of Bayesian networks for on-line learning or tackling the problem using first order learning techniques. Programming by demonstration is interesting because it aims to solve the almost paradoxical problem of current interfaces not facilitating the use of computers for performing repetitive tasks. Additionally, programming by demonstration is best implemented as an autonomous agent system. There is currently a great deal of interest in autonomous agents because they are intrinsically appealing to their human users, forming the most natural form of interface yet invented.

References

[1] Kay A. User Interface: A personal view. In B. Laurel, editor, *The Art of Human-Computer Interface Design*. Adison-Wesley, Reading, Mass, 1990.

[2] Schlimmer J. C. and Hermens L. A. Software agents: Completing patterns and constructing user interfaces. In *Journal of Artificial Intelligence Research*, pages 71–89, 1993.

[3] A. Cypher. EAGER: Programming repetitive tasks by example. In *Proceedings of CHI*, pages 33–39. ACM: New Orleans, 1991.

[4] A. Cypher, editor. *Watch What I Do*. MIT Press: Cambridge MA, 1993.

[5] Davison B. D. and Hirsh H. Predicting sequences of user actions. In *AAAI/ICML 1998 Workshop of Predicting the Future: AI Approaches to Time-Series Analysis*, pages 5–12. AAAI Press, 1998.

[6] N Dahlbäck, A. Jönsson, and L. Ahrenberg. Wizard of oz studies - why and how. In W Gray, W. E. Hefley, and D. Murray, editors, *Proceedings of the*

1993 International Workshop on Intelligent User Interfaces. Association of Computing Machinery, Inc, 1993.

[7] Claxton G. *Hare Brain Tortoise Mind*. Forth Estate Books, Douglas, UK, 1997.

[8] L. A. Hermens and J. C. Schlimmer. A Machine Learning Apprentice for teh Completion of Repetitive Forms. In *IEEE Expert*. 28-33, 1994.

[9] D. Hinkle and C. N. Toomey. CLAVIER: Applying Case-based Reasoning to Composite Part Fabrication. In *Proceedings of the Sixth Innovative Applications of Artificial Intelligence Confernece*, pages 55–62. AAAI Press: Seattle, WA, 1994.

[10] E. Horvitz, L. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The Lumière project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users. In *UAI-98*, pages 256–265, 1998.

[11] K. Lang. NEWSWEEDER: Learning to Filter News. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339. Morgan Kaufmann: Lake Tohoe, CA, 1995.

[12] P. Langley. Machine Learning for Adaptive User Interfaces. In *Proceedings of the 21st German Annual Conference on Artificial Intelligence*, pages 53–62. Springer: Freiburg, Germany, 1997.

[13] P. Langley. User Modeling in Adaptive Interfaces. In J. Kay, editor, *Proceedings of the Seventh Internation Conference on User Modeling*, pages 357–371. Springer, 1999.

[14] H. Lieberman. Integrating User Interface Agents with Conventional Applications. In *Proceedings of IUI 1999*, 1999.

[15] P. Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, pages 31–40, 1994.

[16] G. W. Paynter. FAMILIAR: automating repetition in common application. In *Proceeding of the Third New Zealand Computer Science Research Confernence*, 1999.

[17] M. Pazzani, J. Muramatsa, and D. Billsus. SYSKILL & WEBERT: Identifying interesting web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 54–61. AAAI Press: Portland, OR, 1996.

[18] E. Rich. User modeling via stereotypes. *Cognitive Science*, pages 329 – 354, 1979.

[19] S. Rogers and W. Iba, editors. *Adaptive User Interfaces: Papers from the 2000 AAAI Symposium*. AAAI Press, March 2000. Technical Report SS-00-01.

[20] L. Strachan, J. Anderson, M. Sneesby, and M. Evans. Pragmatic User Modelling in a Commerical Software System. In A. Jameson, C. Paris, and C. Tasso, editors, *Proceedings of the Sixth International Confernece on User Modelling*, pages 189–200. SpringerWienNewYork, 1997.

[21] D. H. Wolpert. Stacked generalisation. In *Neural Networks, Vol. 5*, pages 241–259. Pergamon Press, 1993.