SOME VIRTUES AND LIMITATIONS OF ACTION INFERRING INTERFACES

Edwin Bos

Nijmegen Institute for Cognition and Information (NICI)
University of Nijmegen
PO Box 9104, 6500 HE Nijmegen
The Netherlands
E-Mail: bos@nici.kun.nl

ABSTRACT
An action inferring facility for a multimodal interface called Edward is described. Based on the actions the user performs, Edward anticipates future actions and offers to perform them automatically. The system uses inductive inference to anticipate actions. It generalizes over arguments and results, and detects patterns on the basis of a small sequence of user actions, e.g. "copy a lisp file; change extension of original file into .org; put the copy in the backup folder". Multimodality (particularly the combination of natural language and simulated pointing gestures) and the reuse of patterns are important new features. Some possibilities and problems of action inferring interfaces in general are addressed. Action inferring interfaces are particularly useful for professional users of general-purpose applications. Such users are unable to program repetitive patterns because either the applications do not provide the facilities or the users lack the capabilities.

KEYWORDS: programming by example, demonstrational interfaces, multimodal interfaces.

1 INTRODUCTION

A recent development in human-computer interaction concerns the creation of demonstrational interfaces (e.g., [17]). "Demonstrational interfaces allow the user to create parameterized procedures and other high-level abstractions without requiring the user to learn a programming language. The key feature of a demonstrational interface is that the user performs actions on concrete example objects (often, by direct manipulation), but a more general-purpose procedure is created. The term 'demonstrational' is used because the user is *demonstrating* the desired result using example values" [16, page 11].

Demonstrational interfaces meet the needs of professional users of general-purpose applications such as word processors and drawing programs. These systems do not provide their users ways to express their user-specific high-level tasks, so the users have to decompose them into many atomic actions. Repetition of such high-level tasks thus results in a series of repetitive routine actions, obviously causing a feeling of discontent. In principle, the programming facilities that some applications offer could prevent discontent. By means of writing parameterized macros or simple computer programs the user could compile the atomic actions into a single compound action. Unfortunately, most users lack the capabilities or time required to learn to construct such programs. Dealing with typed variables, control structures, etc. requires abstraction in a field unknown to them. The solution lies in the idea of having the system instead of the user deal with abstraction: *programming by example*, the first name for demonstrational interfaces.

Myers [16] provides a solid framework of most of the demonstrational interfaces known to date. He reviews all possible types of demonstrational interfaces, from the simplest (e.g., Emacs' keyboard macros [19], in which the user can enter macro recording mode, perform actions as usual, stop macro recording mode, and invoke the macro recorded), to the non-intelligent programmable ones (e.g., Smallstar [12], in which the user can edit a mixed text-and graphics representation of a recorded script of actions and add generalizations), on to the more intelligent, programmable ones that infer generalizations from actions (e.g., Peridot [15]; Metamouse [14]; and Eager [8], an interface that automates repetitive tasks in the HyperCard environment and does not require the user to go into a special recording mode). Use of inductive inference distinguishes intelligent from simple interfaces. If inductive inference is used, the system infers from previous actions (the examples) which actions shall probably be executed next.

This paper focuses on demonstrational interfaces with inductive inference capabilities. Since in those interfaces the user need not know nor even be aware of the programming-by-example facilities, I prefer to use the name *action inferring interfaces*. This name reveals the key issue better than *programming by example* and *demonstrational interfaces*. These terms suggest a conscious activity (programming, providing examples) and the involvement of dummies (viz. the examples), which is not the case in inductive inference interfaces.

The goal of this paper is to examine some virtues and limitations of action inferring interfaces. In order to be able to do so, I developed (inspired by Allen Cypher's Eager) an action inferring facility for the multimodal interaction

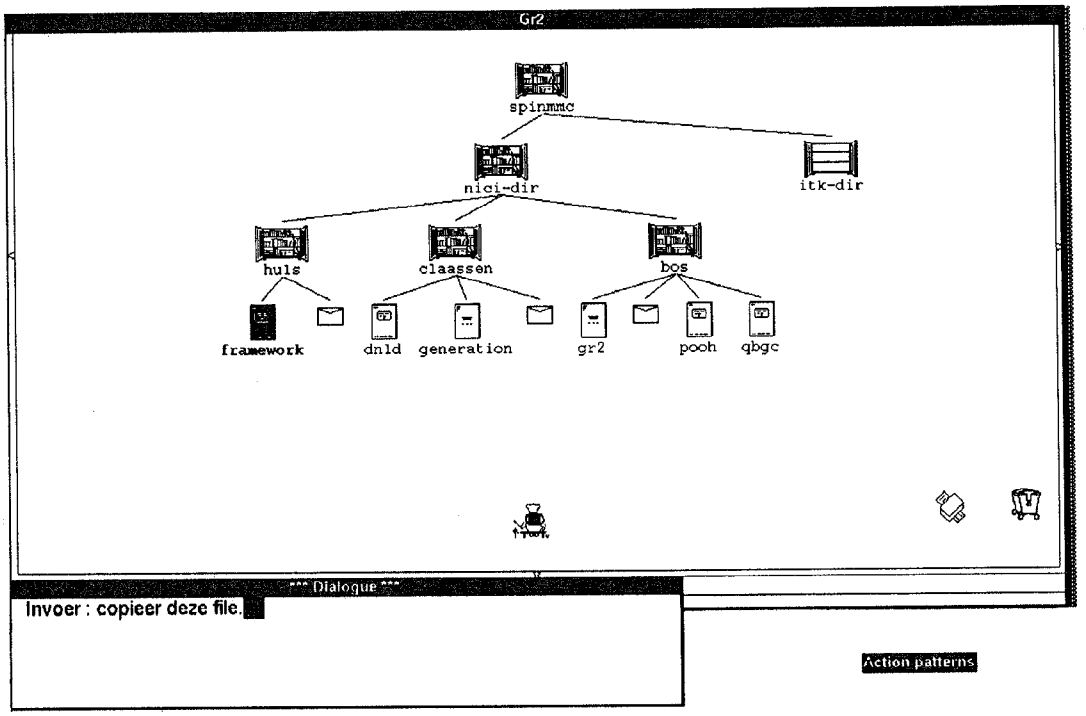
module Edward [2]. I was particularly interested in exploring action inference in a *multimodal* environment. So far, action inference has only been applied in unimodal environments. Edward, written in Allegro Common Lisp and running on a DECstation, combines the positive features of the two modalities of interaction, the *language* and the *action* modality [4]. Edward integrates a graph-editor called Gr^2 [1] and a Dutch natural language dialogue system called DoNaLD [6], coupled by an interaction manager. One of Edward's application domains involves a file system environment with directories, several types of files, authors, a garbage container and so forth. The user can interact with Edward by manipulating the graphical representation of the file system (a directed graph), by menus, by typed, handwritten, or spoken natural language (NL) or command language, or by combinations of these. Edward's output comprises graphics and natural language (written and spoken). Edward's knowledge sources include a semantic network for type information (representing, e.g., that email messages are files, which are computer concepts, which are entities), a database for token information (representing, e.g., that <directory#2> contains <email#7>), and a context model for salience information (representing, e.g., that <directory#2> has just been mentioned and thus can be referred to by "it"). The action inference facility I developed has been incorporated in Edward's interaction manager.

The paper is structured as follows. First I will provide a thorough view on Edward's inductive inference capabilities (functionality and user interface). Next, I will describe in some detail the resources and algorithms Edward uses to anticipate future actions. Then, in order to stimulate discussion and further research, some virtues and limitations of action inferring interfaces in general are given.

2 ACTION INFERENCE IN EDWARD

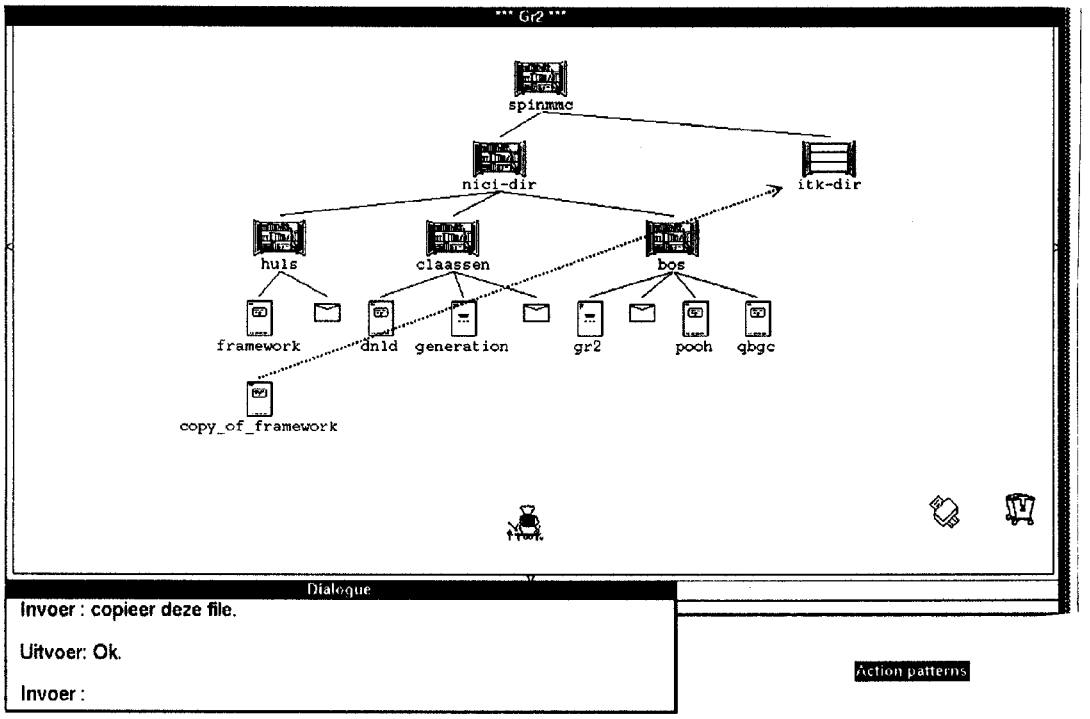
I start the description of action inference in Edward with an extensive example. Figure 1 shows a file system interface, with labeled bookcase icons for directories (e.g., *huls* and the empty *itk-dir*), envelope icons for email messages, labeled report icons (*generation* and *gr2*), slightly different report icons for a special kind of report called spin-report (e.g., *framework* and *dnld*), and a bear icon representing the system. In this example, the user puts copies of all reports into *itk-dir*. In the rest of this section I will describe three different aspects of action inferring interfaces, viz. inference, execution, and management.

2.1 Example I



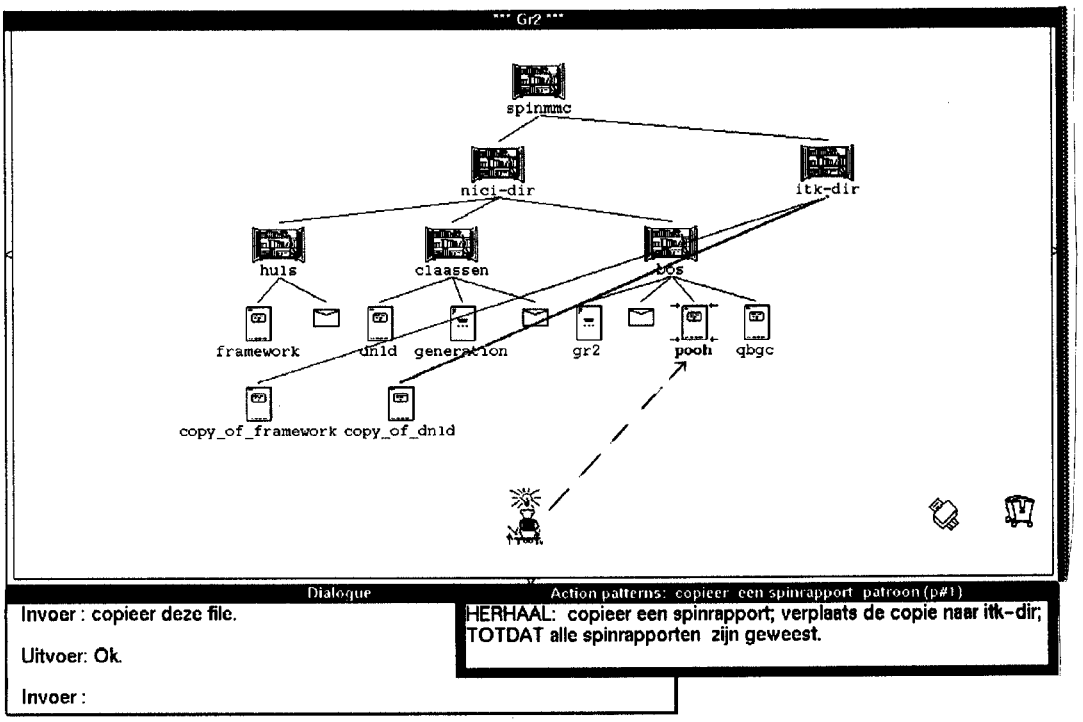
(a)

Fig. 1a. Suppose the first action the user performs is entering the NL command "copy this file" (in the Dialogue window on the lower left) after having selected the spin-report *framework* (Action 1).



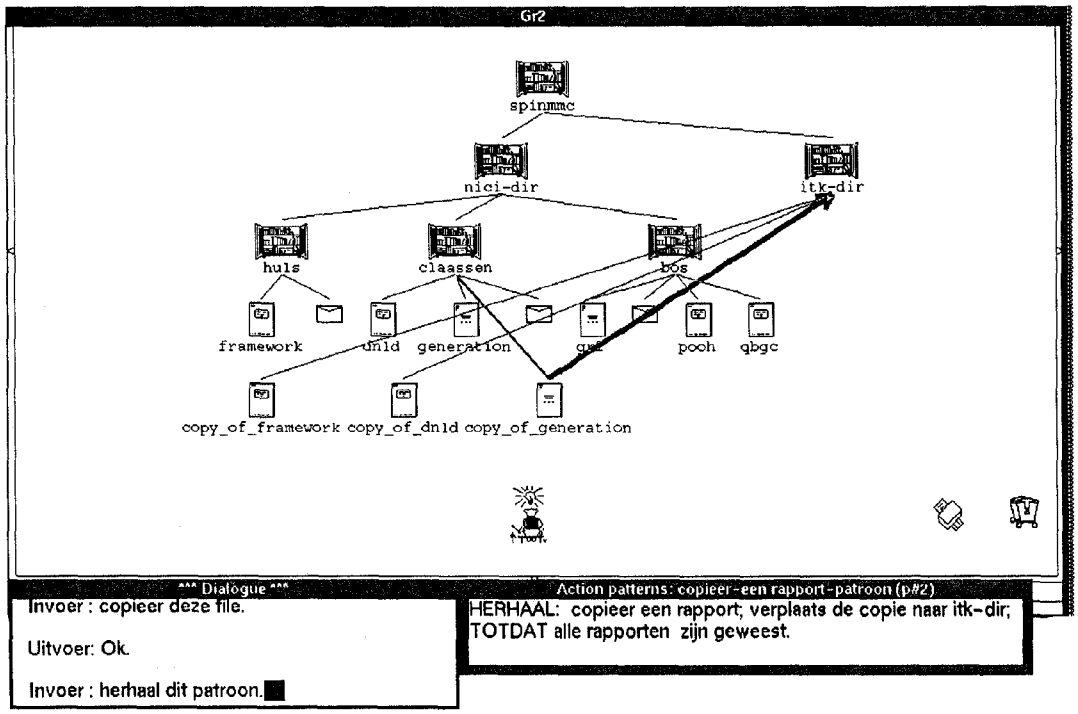
(b)

Fig. 1b. Next, the user moves the copy *copy_of_framework* to *itk-dir* by redirecting the arc from *huls* to *itk-dir* (Action 2).



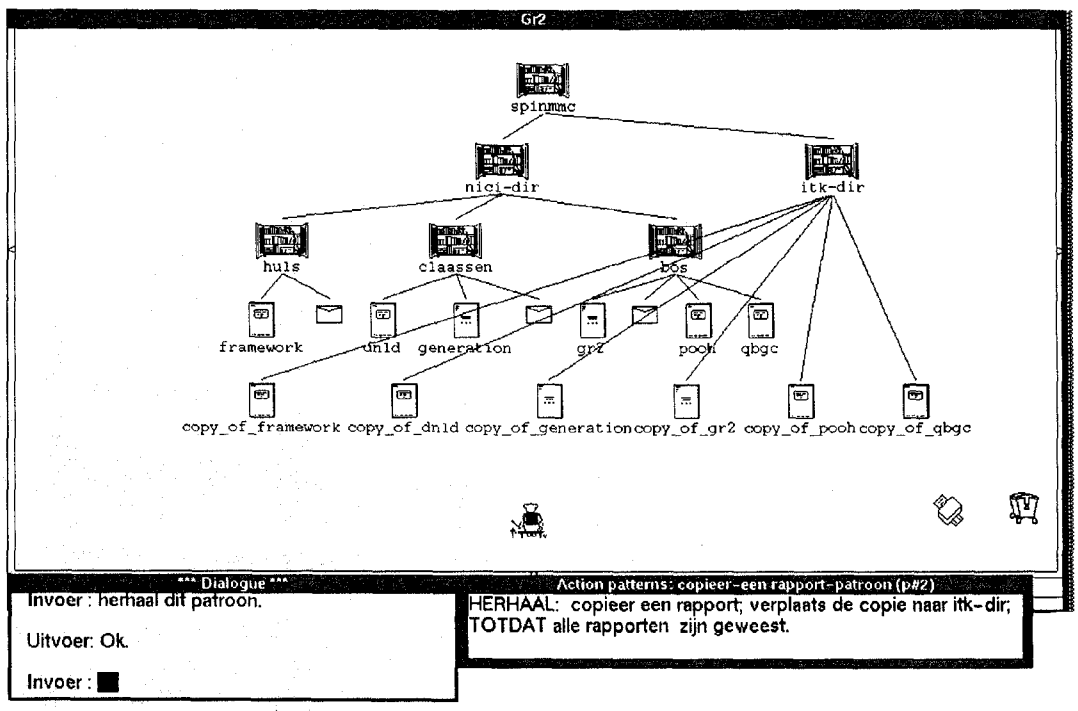
(c)

Fig. 1c. By selecting a menu option, another copy command is applied by the user, this time to the spin-report *dnld* (Action 3); and the copy is redirected to *itk-dir* (Action 4). Upon completion, the system detects a pattern. The next action anticipated is suggested, being the selection of the spin-report *pooh*, and the detected pattern is described in NL in the Action patterns window, saying "REPEAT: copy a spin-report; move the copy to itk-dir; UNTIL all spin-reports have been handled".



(d)

Fig. 1d. The user, however, selects another icon than anticipated, viz. the report *generation* (not a spin-report), and copies it (Action 5). Now the system anticipates a different pattern, this time concerning reports in general instead of just spin-reports, and suggests a graphical redirect of *copy_of_generation* to *itk-dir*. Having seen this pattern description, the user commands Edward to "repeat this pattern" (in the Dialogue window).



(e)

Fig. 1e. Upon the command "Repeat this pattern." (Action 6), all anticipated actions are executed automatically, resulting in six copies of reports in *itk-dir*.

2.2 Inference

Edward constantly monitors the user's actions and signals when it finds a repetition of actions. After 4 actions in Example I, the iterative pattern "copy a spin-report; move the copy to itk-dir" is detected. To indicate its detection ot a pattern, Edward does three things: First of all, to show that Edward 'knows' what's going on, an idea icon appears above the system icon. Secondly, Edward generates a (visual and, optionally, audio) NL description of the pattern detected (see the window in the lower right corner of Fig. 1c). Thirdly, Edward suggests the action it expects the user to execute next. If that action is the selection of an object, the system points at that particular object and the object is marked by small arrows (see *pooh* in Fig. 1c). To simulate the pointing gesture, an arrow grows from the system icon to the icon of the object. This pointing gesture simulation is also used if Edward generates multimodal referring expressions in answers to user questions. E.g., "Stuurde Alice deze email?" "Nee, deze ↗ email stuurde zij." ("Did Alice send this email?" "No, she sent this ↗ email."), where ↗ stands for a pointing gesture to the visual representation of the referent of the concurrent phrase. The object pointed at by Edward remains marked by small arrows as long as the user does not act. If the argument of the next action has already been selected, the anticipated command is suggested, in the same modality as it was given previously. If the previous command was given by menu option selection, the menu automatically pops up and the option corresponding to the command is preselected. If the previous command was given in the action modality, which, for example, is the case in dragging icons and redirecting arcs, a simulated pointing gesture is used. For example, an arrow growing from the current icon position to the expected position, where the icon subsequently is depicted by contours only; or an arrow growing to the expected new start or end node for an arc (see Fig. 1d). If the previous command was given in natural or command language, the expected next action is also described in that modality, e.g., "Verplaats qbgc naar claassen." ("Move qbgc to claassen.") or "cl itk-dir". This description is displayed after the user's input prompt in the Dialogue window.

All these indications of Edward's detection of an action pattern follow the same design goal as was set for Eager: minimal intrusion of the user's actions. And as in Eager, executing an action that matches the anticipation is implicitly confirming; performing any other action is implicitly rejecting.[1] The start and end of action sequences do not need to be explicitly stated.

It will be obvious that two similar actions will quite often be insufficient to exhibit the user's true intentions. In Example I, for instance, Action 5 reveals that not just spin-reports should be moved to *itk-dir*. Hence, based on the last 5 actions, Edward composes the iterative pattern "copy a report; move the copy to itk-dir". It anticipates the selection of the arc representing the contain-relation of *copy_of_generation*, followed by a redirect command. In Edward's Flags window a variable named 'reliability' is available for user adaptation. It controls the number of iterations that must have been performed by the user before Edward starts anticipating. It defaults to 2; there are two cases for which 1 additional iteration is required: singular patterns (i.e., patterns consisting of one action only), and patterns with actions expressed in natural or command language only (because, unlike other modalities, in language the user can express abstraction explicitly).

[1] In the language mode, any other key than *Enter* erases the anticipated action description.

2.3 Execution

Once Edward has detected a repetitive pattern and has shown this by providing anticipation feedback, the user can tell it to automatically execute the inferred actions. The user can give the command to execute inferred actions in three different ways. First the user can click on the system icon. This icon has been dynamically attributed with an idea icon upon pattern detection, and simulated pointing gestures have originated from it, so it is likely to be associated with automatic action execution. The second option is selecting the "repeat pattern"-option from the main menu. The third option is giving a command in the Dialogue window, either in natural language (e.g., "Voltooi de iteratie." ("Complete the loop."); "Herhaal dit patroon!" ("Repeat this pattern!")) or command language ("rp"). In addition, a "next action" and a "next step" command are available, for respectively automatic execution of the next action and the next actions in the current iteration only.

During execution, anticipation feedback is withheld. Action feedback, however, is not withheld: The graphical representation is updated after each action (e.g., by creating a new icon after a copy action). In this way the world changes gradually instead of at once, which makes it easier for the user to update her mental model of the state the system is in.

It is of course of great importance that the user feels confident about Edward's inductive inference capabilities. Confidence is mainly gained by instantiation of the next user actions anticipated by the system and the NL description of the pattern detected. Additional confidence is gained by successful inference in the past (if Edward anticipated a couple of actions correctly, it will anticipate them all). The presupposition that users themselves apply induction was found a valid one in Eager's small user study [8]. However, if recovery from an unsuccessful automatic action execution requires a lot of effort, even confident users will hesitate to have Edward complete the loop. So ease of recovery is crucial. In Edward, the user can give the undo command to return to the situation previous to the execution command. Going back one by one (either per single action or per set of actions in an iteration) through the states that resulted from the automatically executed actions is also possible.

2.4 Management

On some occasions the repetitive pattern of actions might be needed again. Suppose the user of Example I wants to copy some new reports and redirect them to *itk-dir* later on in her session. Of course, she can copy again, resulting in the creation of a new pattern. However, Edward (unlike Eager) also provides her access to the pattern once constructed. Each

pattern is automatically named by Edward. Since automatic naming is a research subject in itself, no effort has been spent to have Edward summarize a pattern into a proper name. Instead the first action is used (e.g., for the first pattern detected in Example I "copy-a-spin-report-pattern"). Shorter aliases are generated as well (e.g., "p#1"). If the user does not like an automatically generated name, she can change it. Pattern names and aliases are communicated to the user in the title bar of the Action patterns window (see Fig. 1c, lower right). Generally, the user will first want to view a pattern inferred in the past, before using it again. This can be achieved by giving commands like "Beschrijf p#1." ("Describe p#1."), upon which Edward displays a NL description of that particular pattern. If its description matches the user's needs, the pattern can be executed again, e.g. by "Herhaal het." ("Repeat it.").

Instead of simple reuse, the user might want to perform a slightly different pattern, e.g., instead of moving copies to *itk-dir*, moving them to *claassen*. Therefore, the user can edit existing patterns. Upon the edit-pattern command, Edward displays a form on which for each action the command, arguments and results are listed (on separate lines). The user can edit these NL descriptions. E.g., changing a constant result such as "itk-dir" into "claassen", or a variable argument such as "spin-reports" into "emails". Changing a constant into variable or vice versa, changing a command into another, erasing an entire action, and other pattern structure editing is not (yet) supported. After a pattern has been edited, both the original and the edited pattern are available for use.

3 EDWARD'S RESOURCES AND ALGORITHMS

3.1 How the actions are inferred

Edward stacks descriptions of all so-called *core actions* the user performs. Edward's undo mechanism makes use of this action description stack. Currently, in the file system domain, the core actions are copy, delete, redirect, move, close, open, format, create, and rename commands. The number of core actions has no effect on the quality of the inductive inference. Note that selection is not stacked, nor are information requests, menu calls, scroll actions, and actions involving patterns (complete, repeat, edit). Hence, such commands do not mess up pattern construction.

Of each core action just completed, a description is pushed onto the stack. Descriptions list the command of the core action, the argument to which the action was applied, the result of the action, if any, and the modality the action was performed in. Upon stacking, Edward tries to detect an action pattern. The modality the actions are performed in is irrelevant to the pattern recognition system. Edward examines the stack and looks for a similar action description. Actions are considered similar if and only if 1) the commands are the same, and 2) the arguments are either of the same type or of the same supertype within a certain *semantic distance* (semantic distance between concept A and B is defined as the number of arcs in the shortest path from A to B in the semantic network). The distance restriction is necessary since all arguments are of supertype <concept>. Hence, because spin-reports and women, for instance, are semantically too remote, renaming *dnld* and renaming *Alice* are not considered similar actions.

If a similar action has been found on the stack, that point in the stack is marked. If new actions pushed onto the stack match the actions following the marked one, the user is assumed to perform the actions of the second iteration of the loop. As soon as enough (i.e. 'reliability' number of) consecutive similar actions have been encountered, Edward announces that it has detected a repetitive pattern. As soon as a new action does not match the pattern (e.g., Action 5 for p#1 in Example I), the stack mark is removed and the pattern is discarded. As Example I shows, the very same action may trigger the immediate creation of a new pattern. This new pattern is created entirely independent from the previous one, i.e., it is not an expansion of its fairly similar predecessor.

Edward generalizes over the arguments of similar actions. It tries to find an assertion valid for all arguments, by following a combination of two rules of induction: the method of *agreement* and the method of *difference*. These rules of induction are the two most commonly used and were formulated by John Stuart Mill (in [10]). Examples of generalizations inferred by Edward are: all arguments are the same (e.g., <directory#2>); all arguments have a direct relationship to the preceding action (e.g., the result of it); all arguments are of the same type (e.g., all spin-reports); all arguments are of the same type and have certain features in common (e.g., all empty directories, all closed directories, all leaves in the directory tree, i.e. not containing subdirectories); all arguments are of the same type and have a particular relation token filler in common (e.g., all persons with a live-in relation with place filler <town#7>); all arguments are of the same supertype within a certain semantic distance (e.g., <email#9>, <spin-report#12>, and <dissertation#2> are all files). The order in which assertions are tested for validity is fixed, and works from specific (identical) to general (same supertype). This order yields conservatism in the patterns generated.

When a pattern has been detected, a program is constructed. Its body consists of statements, each describing a set of similar actions. The statement describing the similar actions 1 and 3 of Example I is (copy (a spin-report)), where (a spin-report) denotes the variable argument. Apart from a command and argument, statements can also contain a result. Move action statements, for example, contain a result describing the positions where to move to. This can be a constant, i.e., a fixed position such as <screen-position#6>, but also a variable, a so-called position pattern. Edward generalizes over positions by laying a grid over the graphical display. Like Eager, Edward recognizes sequences of numeric data with a given tolerance, which in Edward's case depends on the icon width and height. For example, (50,100), (93,-49), (127,-205) is generalized as (50+i×40,100-i×150) with a tolerance of 5. If Edward fails to find an absolute pattern, it tries to find a relative position pattern, for instance, a shift of 40 towards the left. If no position pattern can be found, inference fails.

Each statement within the body of the program is associated with a stop condition. The body of the loop is executed until

one of the stop conditions is satisfied. The stop condition of the statements describing similar actions 1, 3 and 5 of p#2 of Example I is "all reports that are not the result of any copy action within the loop have been copied". The copies themselves are excluded from the loop because otherwise an eternal loop would be created (they would be copied too, etc.). Results of actions must always be excluded. Other cases of exclusion concern action sequences in which the user aligns icons next to a 'base icon'. Suppose, in Fig. 1a, the user would, instead of copying *framework*, want to align all other spin-report icons under *framework* (the base icon). Clearly, she does not want to move *framework*, that icon is already aligned without any action. In order to prevent automatic movement of *framework* towards the end of the column of aligned icons, *framework* must be excluded from the set of arguments of the move action statement of the loop. Exclusion is achieved by matching all icons with the position pattern specified in the move action statement. E.g., if *framework* is at (150,100) and the position pattern is (150,75-i×25), *framework* is excluded because its position matches the position pattern for i=-1.

Apart from a stop condition, an argument order function is also associated with each statement. Examples of argument order functions are "declining x-position" and "alphabetical descending of label". The argument order will be used in execution to determine which of the possible candidate arguments should be selected first (e.g., in Fig. 1c, *pooh* is assumed to be selected next and not *qbgc*). Sometimes no order can be determined by Edward; perhaps the user chose the arguments randomly, but it is more likely that the user applied an ordering criterion unknown to Edward.

3.2 Instantiation and execution

If a pattern has been detected, the next action anticipated is instantiated (e.g., redirect in Fig. 1d). Instantiation is effective because it communicates the pattern in terms of the interface language. The Eager user study showed that users have no difficulties with instantiation. The argument of the next action is determined by its statement, the argument order function associated with it, and all the corresponding arguments already on the action stack. First all possible arguments described by the statement are compiled (e.g., <directory#2>; all reports). Then, in case the statement contains variables, the corresponding arguments and results already on the action stack are omitted, as are base icons, and the argument order function is called. This function selects the first one in the set of candidates. For instance, the declining x-position function yields the first file icon next to the previous one. When no argument order function is specified, a random object is taken from the set. Sometimes the user selects another object than is anticipated by Edward. If this object is a member of the set of possible candidates, Edward simply anticipates the next action for that particular argument. The pattern is not discarded until the user performs an action dissimilar to the actions in the pattern.

Whenever a pattern is detected, Edward creates a pattern token (e.g., <pattern#5>), and brings it into the context model. As long as the pattern can be repeated, it remains in the context. If the user diverges from the pattern, the token is deleted. Using the context model, Edward is able to understand expressions like "Repeat it." and "Complete the loop." in exactly the same way as other referring expressions are understood (a detailed description of referent resolution in Edward is in preparation [5]).

If Edward is requested to execute the program it constructed from the repetitive pattern, it consecutively anticipates the next action, performs it, and stacks it, until a stop condition is satisfied. Before execution, the program is compiled in order to prevent endless loops. Edward checks the body of the program on constants. If all statements concern constants only (e.g., close <directory#2>, and open <directory#2>), the user is prompted for the number of iterations.

The natural language description of the program to be executed is derived from the action pattern. If constants are involved, Edward generates an unambiguous referring expression. Usually this yields the name of the constant (e.g., "itk-dir"), but nameless objects can be described as well (e.g., "Gerard's email"). For a description of the algorithm and resources used to generate referring expressions by Edward, I refer to [3]. If variables are involved, Edward uses the type information stored in the lexicon (particularly the gender information of the lemma associated with the type: "copie" ("copy"), e.g., is a male noun and must be referred to with, e.g., "hem" ("him") or "zijn" ("his")). If position formulae are involved, LISP expressions are used for the description (e.g., (+ (* i 20) 30)).

The NL description is not stored but is generated each time when required (e.g., when the user wants to view a pattern before reusing it). This is done in order to be able to include new information such as names for previously nameless objects, and to deal with possible ambiguities in the new context. For example, another email from Gerard could have come in, making "Gerard's email" ambiguous. Edward will generate a new unambiguous expression, e.g., "Gerard's email about parsing".

NL descriptions are also generated for the form that is displayed if the user wants to edit an existing pattern. For arguments and results, these descriptions are noun phrases, e.g., proper noun phrases such as "itk-dir", or indefinite descriptions such as "een spinrapport" ("a spin-report"). The descriptions edited by the user are interpreted by Edward's NL analysis component. If any ambiguities remain after analysis, Edward will ask the user for further clarification. E.g., "does 'Gerard's email' refer to 'Gerard's email about parsing'?". The interface style form-fill in is preferred over free editing of the NL description in the Action patterns window in order to prevent modifications that are not supported by either the action inferring facility or the NL component.

An editable LISP program, displayed at the user's request, is also directly derived from the pattern. This LISP-notation is provided in order to offer, in the future, possibilities to alter the control structure of the loop, which is not possible in the NL description. Currently, however, only the editing of variables and constants is supported; adding conditionals is not yet supported, though implementation seems feasible. Eventually, a scripting language should replace LISP.

4 SOME VIRTUES OF ACTION INFERENCE

The most important virtue of action inferring interfaces such as Edward is that they can reduce user effort and prevent annoyance caused by tedious manual effort. To get a better understanding of how inductive inference works, let us take a look at Norman's Theory of Action [18]. His model distinguishes seven stages in the user's task performance. "The primary, central stage is the establishment of the goal. Then to carry out an action requires three stages: forming the intention, specifying the action sequence, and executing the action. To assess the effect of the action also requires three stages [..]: perceiving the system state, interpreting the state, and evaluating the interpreted state with respect to the original goals and intentions" [18, page 42]. The model has to be interpreted recursively, that is to say, intentions are subdivided into subintentions, etc., until they are atomic and can be dealt with in the next stage.

If we apply the model to the interactions of Example I, we see that the user subdivides her compound[2] intention "copy all reports to itk-dir" into the individual atomic intentions "copy framework", "move copy", etc. The activities of the system can be divided over the same seven stages as Norman distinguishes for the user. In fact, all human-computer interactions can be integrated into one model, where the user's output at the physical level equals the system input at that level, and where the system output is the user's input. Moreover, both interactors maintain a model of the other interactor's activities. In the literature these models usually are referred to as a conceptual model (maintained by the user) and a user model (maintained by the system). Inductive inference means having the system construct a representation of the user's compound intention on the basis of a sequence of user signals. The systems' representation of the user's compound intention is then mapped onto the system's own compound intention, which, at user request, is subsequently subdivided into atomic intentions, which are formulated and executed by the system in succession.

If the 'inference' effort, that is the sum of the effort the user must spend to understand the inference of the system and to command the system to automatically perform the actions inferred, is less than the effort the user must spend to execute the actions herself, then inductive inference is worthwhile. This is typically the case with repetitive actions. Another type of situation in which inductive inference can be realized is when a sequence of actions is performed that is typical for the domain. User studies could provide the knowledge about ways to achieve certain goals in the domain; this knowledge could be incorporated in the inference mechanism. If goal g typically is achieved by intention sequence $i_1,..,i_n$, this knowledge could be incorporated and used when a particular user has executed some actions corresponding with the start of the sequence. Links with intelligent help facilities are obvious: If g can also be achieved by one single intention i, the system could suggest the use of i next time. The incorporation of error scripts, that is, action sequence descriptions that lead to typical errors, could enable the system to prevent the user from entering error paths too deeply and to suggest correct paths.

[2] In fact, it need not be compound: the user could also have formulated her intention in one single natural or formal language expression.

5 SOME PROBLEMS OF ACTION INFERENCE

5.1 Inductive inference

It is known from philosophy that inductively derived rules cannot be proven correct [10, page 4]. Therefore, it will come as no surprise that there are a couple of fundamental problems of inductive inference in the interface. First of all, there is no guarantee that the actions that are input to the inductive inference mechanism are part of a sequence. These actions might be the only ones the user wants to execute, that is, she might not want to iterate at all.

Secondly, noise can arise because deep-structure intentions are induced from surface-level actions. Users can make errors in the execution of an action, and in the translation of their intentions, both resulting in another action; they also can form inappropriate intentions (respectively called slips and mistakes by Lewis and Norman [13]). Moreover, other intentions which cannot be reasonably expected may exist: the user might, for instance, not at all intend to align file icons while dragging, but intend to make some space available by moving the objects, which, by sheer luck or by motoric constraints, end up in a line.

Thirdly, there is the problem of overgeneralization. For example, a user may copy only reports of people who are present in her room at that particular moment. Of course, many patterns can be detected (enough to justify action inferring interfaces in my opinion), both domain independent ones and domain dependent ones. Perfect pattern detection (i.e. individual user and situational independent), however, is in principle impossible, even if systems are equipped with additional information channels such as vision systems and full knowledge of the world. Even then, copying the reports of the people who were present at the user's birthday party last night could not be detected. This problem of overgeneralization, by the way, affects people too, "but - unlike machines we people have a background of common sense: expert knowledge against which the limitations of such a potential overgeneralization can be tested" [10, page 9].

5.2 Anticipation feedback

The aim of anticipation feedback is to make clear which actions are inferred by the system. Three options are available, each with their own problems.

1) Instantiation

This option was invented by Cypher [8] and is adopted in Eager and Edward. It shifts the burden of abstraction onto the user. This causes a termination problem: Instantiation leaves determination of when the loop terminates open to the user. Sometimes she might determine termination wrong, so she will end up with a prematurely terminated loop or with a loop continuing too long. An additional problem is that in action mode interfaces, instantiation sometimes violates the design goal of minimal intrusion. If, for example, the next action involves an argument outside the current viewport, reference will be difficult.

2) Generalization
This option is adopted in Edward (twice: the NL and the LISP description). Generalization is problematic because the information to be conveyed is abstract, while the users are assumed to have difficulties in producing abstractions. A NL description is a way of generalization in which abstraction is dealt with in the user's own language (e.g., "all reports", "every lisp file of Gerard in the backup directory"). Readability problems arise if the patterns to be described get more complex. First, lengthy patterns imply lengthy descriptions, careful study of which may require even more user effort than needed for the manual execution of the actions. Secondly, the incorporation of conditionals reveals the ambiguity problem of NL: Use of several different conjunctions results in incomprehensible descriptions flooded with *and*s and *or*s. Furthermore, NL descriptions are not necessarily interface language expressions. Reference to non-linguistic interface concepts such as positions and areas, for instance, is therefore problematic. Besides, it is important to note that a NL description requires a user model. Using the system's internal model instead could yield incorrect descriptions, as for instance Cypher found out in Eager which produced "copy every other card" where "copy every card" was appropriate [9]. Maintaining a perfect user model, however, is still problematic.

3) Final state description
This option was chosen in a preliminary implementation of Edward's action inferring facility. It may be seen as a combination of the first two options: instantiation of all the actions anticipated. When dealing with a graphical representation, this option often results in information overflow, especially if objects outside the current viewport are involved. When dealing with language, complex or lengthy iterations result in lengthy descriptions in which missing or undesired actions are difficult to notice.

Although Edward's multimodal environment provides a solution to some extent (by combining instantiation and generalization), a complete solution to the anticipation feedback problem is still far away. It is obvious that multimodality is a required feature: language for providing abstraction, and graphics for referring to non-linguistic interface concepts. Multimodal interfaces with knowledge about the way people access information, and with an algorithm to decide which option to choose in individual cases (depending on the type of actions, the capabilities of the user, etc.), might provide a solution. Initial steps in this direction have already been taken (e.g., [7]).

6 CONCLUSIONS

In this paper I have described an action inferring interface facility for the multimodal environment Edward. The inductive inference method used is domain-independent. The facility has two important features, new in action inferring interfaces. The first important feature is multimodality: the use of NL combined with simulated pointing gestures facilitate the user's decision in making use of the action inference facility. NL provides a language for abstraction, it enables the system to convey information about the termination of the loop. The second important feature is reuse of patterns (possibly edited). Here, too, NL plays a major role.

To test the usefulness, a usability study is needed. However, there are no advanced/expert users of Edward available who are unfamiliar with the inference facility. A pilot study has been conducted with novices, unfamiliar with both Edward and action inferring facilities. These users quickly gave commands for automatic action execution. However, they never used the pattern management facilities. A full usability study of Edward, including the action inferring facility, is planned for the near future. I have no reasons to expect that the outcome with respect to usefulness will be worse than the Eager user study [8], which revealed a high degree of user satisfaction. In fact, I expect to find an even higher degree of satisfaction, due to the availability of NL descriptions and pattern reuse.

Based on the experiences with Edward, I have listed several virtues and limitations of action inferring interfaces in general. Although these lists are by no means exhaustive but rather tentative, I conclude that action inferring interfaces are valuable contributions to the field of human-computer interaction. Since the problems with inductive inference, though fundamental, have not stopped us from using it in daily life, I believe that the problems with action inferring interfaces will not stop users from using them.

Acknowledgements
This research was carried out within the framework of the research programme *Human-Computer Communication using natural language* (MMC). The MMC programme is sponsored by SPIN Stimuleringsprojectteam Informatica-onderzoek, Digital Equipment B.V., BSO, and Sun Microsystems B.V. Thanks to Allen Cypher for inspiring me and for discussing several issues of action inferring interfaces with me. Additional thanks go to Wim Claassen, Alice Dijkstra, and Nick Terhorst for their comments on the manuscript.

7 REFERENCES

1. Bos, E. (in press). A multimodal syntax-directed graph-editor. In L. Neal & G. Szwillus (Eds.) *Structure-based Editors and Environments*. New York: Academic Press.
2. Bos, E., Claassen, W., & Huls, C. (forthcoming). Edward: a multimodal interface. SPIN/MMC research report. Nijmegen: NICI.
3. Claassen, W. (1992). Generating referring expressions in a multimodal environment. In: Dale, R., Hovy, E., Rösner, D., & Stock, O. (Eds.) *Aspects of automated natural language generation*. Proceedings of the 6th international workshop on natural language generation, Trento, Italy, April 5-7. Berlin: Springer, pp. 247-262.
4. Claassen, W., Bos, E., & Huls, C. (1990). The Pooh Way in human-computer interaction: towards multimodal interfaces. SPIN/MMC Research report #5. Nijmegen: NICI.
5. Claassen, W., Bos, E., & Huls, C. (submitted). Automatic referent resolution in deictic and anaphoric expressions. Submitted to *Computational Linguistics*.

6. Claassen, W. & Huls, C. (1991). DoNaLD: A Dutch Natural Language Dialogue system. SPIN/MMC research report #11. Nijmegen: NICI.

7. Conati, C. & Slack, J. (1992). Accessing information through graphics. ECAI'92 Conference proceedings, Vienna, August 3-7.

8. Cypher, A. (1991). Eager: programming repetitive tasks by example. In: Robertson, S.P., Olson, G.M. & Olson, J.S. (Eds.) *Reaching through technology*. CHI'91 Conference proceedings, pp. 33-39.

9. Cypher, A. (1992). Personal communication.

10. Forsyth, R. (Ed.) (1989). *Machine learning*. London: Chapman and Hall.

11. Frohlich, D.M. (1991) The design space of interfaces. In: Kjelldahl, L. (Ed.) Multimedia: principles, systems and applications. Berlin: Springer-Verlag.

12. Halbert, D.C. (1981). *Programming By Example*. PhD Thesis. Computer Science Division, Dept. of EE&CS, University of California.

13. Lewis, C. & Norman, D. (1986). Designing for error. In: Norman, D.A. & Draper, S.W. (Eds.) *User centered system design: new perspectives on human-computer interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

14. Maulsby, D.L., Witten, I.H., & Kittlitz, K.A. (1989). Metamouse: specifying graphical procedures by example. Proceedings of the SIGGRAPH'89, July 1989, Boston, MA.

15. Myers, B.A. (1988). *Creating user interfaces by demonstration*. Boston, MA: Academic Press.

16. Myers, B.A. (1990). Demonstrational interfaces: a step beyond direct manipulation. In: Diaper, D. & Hammond, N. (Eds.) *People and computers VI*. Cambridge: Cambridge University Press.

17. Myers, B.A. (Ed.) (1991). Demonstrational interfaces: coming soon? In: Robertson, S.P., Olson, G.M. & Olson, J.S. (Eds.) *Reaching through technology*. CHI'91 Conference proceedings, pp. 393-397.

18. Norman, D.A. (1986). Cognitive engineering. In: Norman, D.A. & Draper, S.W. (Eds.) *User centered system design: new perspectives on human-computer interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

19. Stallman, R.M. (1979). Emacs: the extensible, customizable, self-documenting display editor. Technical report #519. Cambridge, MA: MIT AI Lab.