These Aren't the Commands You're Looking For: Addressing False Feedforward in Feature-Rich Software

Benjamin Lafreniere[*†], **Parmit K. Chilana**[*], **Adam Fourney**[*], **Michael A. Terry**[*]

[*]University of Waterloo, Ontario, Canada
{*bjlafren,pchilana,afourney,mterry*}@uwaterloo.ca

[†]University of Saskatchewan, Saskatchewan, Canada
ben.lafreniere@gmail.com

ABSTRACT

The names, icons, and tooltips of commands in feature-rich software are an important source of guidance when locating and selecting amongst commands. Unfortunately, these cues can mislead users into believing that a command is appropriate for a given task, when another command would be more appropriate, resulting in wasted time and frustration. In this paper, we present *command disambiguation techniques* that inform the user of alternative commands *before*, *during*, and *after* an incorrect command has been executed. To inform the design of these techniques, we define categories of *false-feedforward errors* caused by misleading interface cues, and identify causes for each. Our techniques are the first designed explicitly to solve this problem in feature-rich software. A user study showed enthusiasm for the techniques, and revealed their potential to play a key role in learning of feature-rich software.

Author Keywords

Feedforward; learning; feature-rich software; help; tooltips

ACM Classification Keywords

H.5.2 [User Interfaces]: Graphical user interfaces (GUI).

INTRODUCTION

When using software to perform an unfamiliar task, a common strategy is to scan through menus and toolboxes, searching for relevant functionality [1, 11, 21]. While this approach can lead the user to desired functionality, it can also lead to difficulties due to *false feedforward* provided by the interface. Feedforward (as opposed to *feedback*) informs the user about what the result of an action will be before it has been performed [5, 25]. *False feedforward* occurs when impoverished or misleading cues in the interface cause the user to believe a command or tool is relevant to the task at hand, when it actually is not. This can result in wasted time and errors when using software.

As one example of a false-feedforward error, the GIMP image editor provides two commands named *Flip Vertically*,

UIST '15, November 08 - 11, 2015, Charlotte, NC, USA
Copyright is held by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-3779-3/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807442.2807482

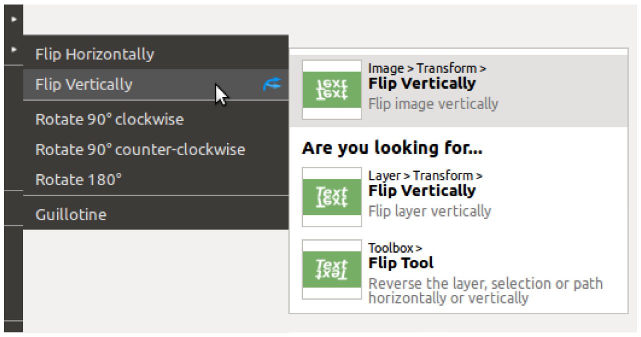


Figure 1. A *Detour Submenu* for the Image > Transform > Flip Vertically menu item in GIMP.

one located in the *Image > Transform* menu, the other in the *Layer > Transform* menu. Though identically named and conceptually similar, these two commands operate on different objects in the document. We have frequently observed novice users of GIMP confusing these two commands, leading to long and frustrating error-recovery periods. Examples of false feedforward of this type can also be found in Microsoft Word and PowerPoint (see Figure 2), and other popular feature-rich applications.

One way to address these errors is to provide better names or icons for commands. For example, the two *Flip Vertically* commands could be renamed to *Flip Layer Vertically* and *Flip Image Vertically* to more clearly indicate the effects of each. However, it is unlikely that false-feedforward errors can ever be completely resolved through the careful design of names, icons, or tooltips – each user brings with them a unique background that will cause them to interpret and understand names and icons differently than other users. Thus, it is worthwhile to develop techniques explicitly designed to help users avoid errors caused by false feedforward in feature-rich software.

With the above as motivation, we created a set of *command disambiguation techniques* to address false feedforward errors in feature-rich software. These techniques: a) provide awareness that a given command can be mistaken for other commands, and b) enable the user to explore, understand, and directly access alternative commands.

Collectively, our prototype designs provide support *before*, *during*, and *after* execution of an incorrect command. As an example of one technique, *Detour Submenus* (Figure 1) display alternative commands in a visually distinct extended menu, where they can be directly invoked.

The specific contributions of this paper are as follows:

- We introduce the concept of *command disambiguation techniques* designed to aid users in detecting and recovering from false-feedforward errors.
- We describe a design space for these techniques, centered on *when* disambiguation occurs and the type of commands involved.
- We present a range of command disambiguation techniques across the design space. Specifically, we demonstrate *Detour Submenus* and *Detour Tooltips* that list potential alternatives to items in menus or toolboxes; *Dialog Link-Ups* and *Tool Hot-Swaps*, which allow users to switch between commands during use; and *Or-do*, a variation on undo that substitutes alternatives for the most-recently executed command.
- We present a qualitative user study that showed enthusiasm for the techniques and revealed their potential to play a key role in learning of feature-rich software.
- Building on Norman's taxonomy of human errors [20], we define two novel categories of error caused by false feedforward – *false-feedforward mistakes* and *false-feedforward slips* – and discuss factors contributing to each.

In the rest of this paper, we start with a review of related work. We then present a conceptual discussion of false feedforward and a design space of techniques to address this issue. Next, we present the prototype designs that we have developed. We conclude with the results of a qualitative user study and a discussion of directions for future work.

RELATED WORK

This work builds on existing research into contextual help; resolving ambiguity in user interfaces; techniques for exploring alternatives; and command recommender systems. We discuss each area below.

Contextual Help

A number of techniques have been proposed to offer guidance directly in the interfaces of applications [2]. Examples such as balloon help [6], ToolClips [8], and enhanced previewing mechanisms [23] assist the user with understanding how particular commands work. Other techniques, such as Stencils [10], demonstrate how to perform multi-step tasks. These techniques can help to prevent false-feedforward errors by providing richer information about individual commands. However, this alone may not be enough to prevent a false-feedforward error, because users often choose the first command that appears to be appropriate for the task at hand, without considering that alternatives may exist.

Contextual help is sometimes used to call attention to alternative commands. In Photoshop CS6, when changing the color space of an image from RGB to Grayscale, a confirmation dialog indicates that the user can control the conversion through use of the separate *Image > Adjustments > Black & White* command. We build on this idea, and explore a design space of techniques for presenting and exploring alternatives for a given command.

Resolving Ambiguity

Past work has developed interaction techniques to resolve ambiguity in the interpretation of low-level input (e.g., gestures, speech) [15], and ways to deal with uncertainty in interpreting and disambiguating input at the level of interface toolkits and frameworks [16, 22]. Our work also addresses problems of uncertainty in user interactions, where the uncertainty arises due to false feedforward cues. In the context of this prior work, the act of choosing a given command could be modeled with a probability distribution related to the likelihood of the command being chosen by the current user as a result of the available cues. The methods developed for these existing toolkits could thus provide a backend for disambiguation techniques that enable the user to verify their choice of a command. Our command disambiguation techniques draw inspiration from this prior work, and examine designs for presenting and enabling the exploration of alternative commands in feature-rich software.

Exploring Alternatives

A number of techniques have been developed to enable users to explore alternatives in user interfaces. Perhaps the most widely used methods for this in feature-rich software are Undo, Redo, and history mechanisms (see [19] for a current review of work on Undo mechanisms). One of our techniques, *Or-do*, builds on this familiar model to allow users to explore alternatives to a command just executed.

Other approaches to support exploration of alternatives have been investigated as well. Subjunctive interfaces enable users to branch a document and then work on multiple alternative scenarios in parallel [14]. Interaction techniques have also been developed for generating and exploring sets of alternative solutions in the domains of graphic design [23, 24], 3D rendering and animation [17], and interaction design [9]. The interaction techniques presented in this paper draw inspiration from this past work, but are designed to enable the exploration of alternatives generated based on likely false feedforward errors. This impacts the design of both where and how alternatives should be presented to the user, because the intent is to prevent or limit the impact of errors, rather than to support free-form exploration of alternatives.

Command Recommender Systems

A number of systems have been investigated for recommending commands that a user could benefit from learning. For example, OWL [13] and CommunityCommands [18] model the user's knowledge of commands, and compare it with similar users in an organization or community in order to make recommendations. The main focus of these efforts has been on making good recommendations, with less attention paid toward investigating different scenarios in which recommendations could be made, or interaction techniques for doing so. The work presented in this paper looks at the latter question, and more specifically how techniques for recommending commands could be used in a targeted way to prevent or ameliorate errors caused by inadequate or misleading feedforward.

In the next section we describe two types of false-feedforward errors that can occur in feature-rich software, and discuss factors that can lead to these types of errors.

FALSE FEEDFORWARD

The concept of *feedforward* was introduced to the HCI community by Djajadiningrat et al. [5], and refers to information provided in an interface to indicate what the *result* of an action will be, before it has been performed. This is closely related to the concept of *perceived affordances*, which refers to the properties of an object that indicate action possibilities (e.g., whether a widget can be clicked), but by most definitions does not address the user's perception of what the results of an action will be.

We define *false feedforward* in analogy to Gaver's concept of *false affordances* [7]. While false affordances are cues that suggest to the user that an action possibility exists, when it actually does not, false feedforward is information that suggests that an action will yield a particular result, when in fact it will not.

False feedforward can cause the user to execute a command that is not appropriate for their current goal. We refer to this scenario as a *false-feedforward error*. In the next section, we distinguish two different types of false-feedforward errors that can occur.

False Feedforward Errors – Slips and Mistakes

Norman distinguishes between two fundamental categories of human errors: *slips* and *mistakes* [20]. Slips occur when an error is made during subconscious actions performed to satisfy a goal, whereas mistakes result from errors in conscious deliberations, such as setting an inappropriate goal. For example, if I invoke the *Image > Transform > Flip Vertically* command with the belief that it will flip a single layer of my document, I have made a mistake. But if I intend to use the correct *Image > Transform > Flip Vertically* command, and accidentally open the *Layer* menu and execute *Layer > Transform > Flip Vertically* instead, it is a slip.

As the preceding example suggests, false feedforward can cause both slips and mistakes. *False-feedforward slips* occur when a user knows the appropriate command to use to reach their goal, but the feedforward the interface provides for another command causes them to invoke that command instead, without realizing that they are doing so.

This may occur when commands with similar appearance or names are visible on the screen at the same time. For example, in Microsoft Word, the commands for working with Comments and Changes in the Review ribbon both include a set of three commands (Delete, Previous and Next; and Reject, Previous, and Next) that are similar in terms of their names, icons, and the actions that they perform (Figure 2 top). A similar situation occurs in the Format tab in PowerPoint, between the tools for adjusting the appearance of shape and text objects (Figure 2 bottom). These similarities can easily lead to false feedforward slips (and often have for the authors of this paper).

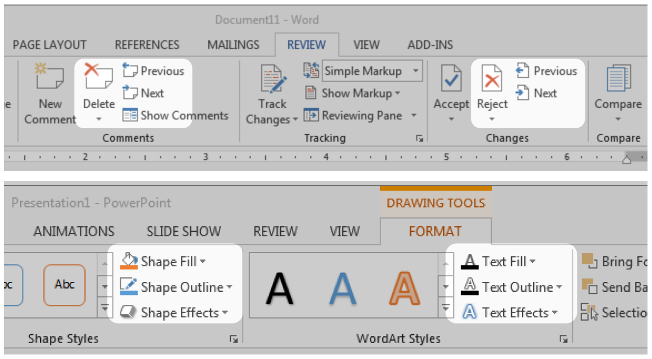


Figure 2. Sets of commands with similar appearances and layout on the Review ribbon in Microsoft Word (top) and the Format ribbon in PowerPoint (bottom).

Once realized, false-feedforward slips are typically easy to recover from, because the user is aware of the correct command to use and thus can easily diagnose the problem. However, this kind of error can still cause frustration by breaking the flow of interaction with the system and forcing the user to engage in corrective actions.

In contrast to slips, *false-feedforward mistakes* occur when the available cues cause the user to invoke a command in the belief that it can satisfy their goal, when in actuality it cannot. In reviewing videos of people performing tasks in GIMP (discussed below) we frequently observed this type of error. This is where we observed our recurring *Flip Vertically* example, which serves to demonstrate the challenge of false feedforward mistakes. Upon making this mistake, users often responded by undoing the operation, then proceeding to reselect the layer they wished to flip, and invoking the incorrect *Image > Transform > Flip Vertically* again. In some cases it took several failed attempts before the user gave up and continued to search for a more appropriate command.

We speculate that because the user is not aware that other commands exist, and because they have formed a belief that the command they have found will satisfy their goal, they are more likely to attribute the unexpected results to other factors than the command they have chosen. As a result, it can take time for the user to revise their belief and begin the search for alternative commands. The techniques we present later in this paper address this by explicitly indicating that alternative commands exist for a given command.

In this paper we mainly look at how interaction techniques could be designed to address false-feedforward mistakes, though some of our designs could help with slips as well.

CAUSES OF FALSE FEEDFORWARD IN FEATURE-RICH SOFTWARE

As a starting point for the design process, we reviewed video of 16 people performing unfamiliar tasks in GIMP, in the context of a previously-published research project [11]. We also inspected the interfaces to GIMP, Microsoft Word, and PowerPoint, and considered our own experiences using all of these applications. Our analysis revealed four specific causes of false feedforward errors in feature-rich software.

Few and impoverished cues. First, the extent of feedforward for most commands in GIMP is provided by each command's name, icon, and tooltip. This provides the user with little information to use when judging whether a command will satisfy a particular goal. Some commands provide richer feedforward mechanisms, such as a live previews of the result of applying a filter, but this is less common.

Cosmetic similarities. Second, the meager feedforward provided by two or more commands can be very similar – the commands may have similar icons, or names that use similar (or identical) terminology. For example, if one does not consider their locations in the menu hierarchy, the two *Flip Vertically* commands in GIMP appear to be identical, albeit with slightly different tooltips. Similarly, in each of the Microsoft Office examples shown in Figure 2, the pairs of commands have very similar icons and names; if the subsection in which each pair of commands is located is not considered, they could easily be mistaken for one another.

Conceptual similarities. Third, commands are often conceptually similar and related, making it difficult for a user with an incomplete understanding of the application or problem domain to determine whether a particular command is appropriate. For example, a new user of GIMP may not have a full understanding of the differences between canvases, images, and layers, and thus might confuse the *Canvas Size, Scale Image,* and *Scale Layer* commands with one another. While all of the operations convey the notion of resizing *something*, each resizes a different type of object in the document and the distinction may not be clear to a new user.

Confirmation bias may play a role here as well, causing users to form an incorrect belief that a particular command is appropriate. For example, upon opening the Canvas Size dialog, the user is presented with parameters for setting width and height, which could be interpreted as a confirmation that they have found the correct command for scaling an image.

Separation in the interface. Finally, the above causes are compounded when similar commands are separated from one another in the interface. For instance, similarly named commands may be found in different top-level menus, toolbars, or ribbons. As a result, the user searching for a command to use may find one of the commands, and, barring any additional knowledge (such as the existence of the other commands), may assume that the first command found represents the correct operation to reach their desired goal. In contrast, if the commands were presented side-by-side, the user would be able to more effectively compare and contrast the available choices, and make a more informed decision about what command is appropriate for the current situation. For example, in Microsoft Word, the Borders dropdown on the Home ribbon could be confused for a command for inserting a table into a document, since no other visible command more clearly affords this operation (Figure 3).

We note that the separation of items in the interface is more likely to contribute to false-feedforward mistakes; as mentioned earlier, commands with similar appearance presented on the screen at the same time can be the cause of false-feedforward slips.

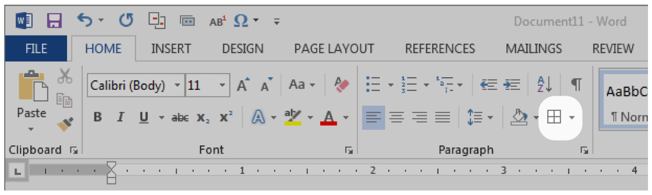


Figure 3. The Borders dropdown on the Home ribbon could be interpreted as a command for inserting tables into a document.

In addition to the above causes, we observed that false-feedforward mistakes occur frequently when users adopt a trial-and-error strategy for performing tasks. This has been observed in previous studies of problem-solving strategies in feature-rich software as well, where "evident, hidden, and false affordances" were identified as the most common factor affecting success in trial-and-error problem-solving episodes [21] (note that the use of the term "false affordances" in [21] corresponds to our definition of false feedforward).

Addressing False Feedforward Errors

Given our characterization of the causes of false feedforward, we can consider two primary approaches to addressing false-feedforward errors.

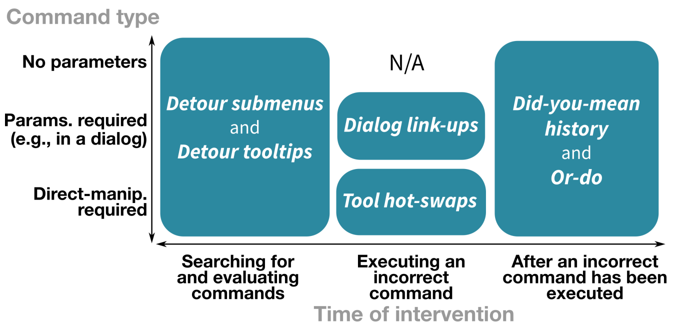
One approach is to address these problems during the design phase, by choosing terms and presentations of commands that reduce the chance of confusion. This approach can reduce the likelihood of misunderstandings due to cosmetic similarities. However, it may be difficult to use this strategy to completely eliminate *conceptual* misunderstandings for commands that have similar effects. For example, the two *Flip Vertically* commands in GIMP both have the effect of vertically flipping something in the interface. Communicating the effect of these operations while also effectively conveying the fact that they flip different things can be challenging; even if the names were changed to *Flip Image Vertically* and *Flip Layer Vertically*, users may not notice the subtle difference, or may be unaware that a more suitable alternative exists in the interface.

The second way to help users deal with false-feedforward errors is to provide interaction mechanisms that enable the user to identify and resolve these potential mistakes. In the sections that follow, we focus on this latter strategy.

A DESIGN SPACE FOR COMMAND DISAMBIGUATION

To address false-feedforward errors, we consider a design space for *command disambiguation techniques*, or mechanisms designed to assist users in a) recognizing that potentially more relevant commands exist, and b) choosing the correct command among a set of alternatives.

Our design space is made up of two dimensions. The first dimension represents *when* an interaction mechanism could assist the user in finding a desired command – before, during, and after they have executed an incorrect command. The second dimension represents the type of command that is being invoked – commands without parameters, commands that require parameters (e.g., entered into a dialog), and direct-manipulation tools. We developed a set of four different techniques covering this design space (Figure 4).



Command type
No parameters
Params. required (e.g., in a dialog)
Direct-manip. required
Detour submenus and Detour tooltips
N/A
Dialog link-ups
Tool hot-swaps
Did-you-mean history and Or-do
Searching for and evaluating commands
Executing an incorrect command
After an incorrect command has been executed
Time of intervention

Figure 4. A design space for addressing false feedforward errors, with the four techniques we have developed located within it.

In developing these designs, we applied four design desiderata that we developed for command disambiguation mechanisms:

Don't interrupt the user's work. False-feedforward mistakes will become less common as the user gains more experience with the application. Thus, it is desirable for disambiguation mechanisms to be as unobtrusive as possible – they should be available when needed, but not burden experienced users.

Provide assistance throughout the interface. Because these errors can often occur during trial-and-error problem solving, techniques should present information in a way that can naturally be found as users explore the interface. Furthermore, because users will not know when they are about to commit an error, any intervention mechanism should be available in a place close to where the user is likely to make such an error.

Teach the user the correct alternative. To reduce the likelihood of the user repeatedly committing the same mistake, the system should not only provide a means of recovering from a false feedforward error, but should also teach users the correct command. In part, this goal is designed to address conceptual misunderstandings by teaching users the appropriate concepts and domain knowledge needed to effectively use the application.

Enable efficient and flexible exploration of alternatives. It takes time and cognitive resources to evaluate alternatives. Thus, whenever possible, the system should help save the user effort when exploring alternative commands. The system should also be flexible in *when* it enables the user to choose an alternative command, so the user can feel free to try out alternative commands (or to ignore them) in the moment, secure in the knowledge that there will be other opportunities to explore them at a later time.

COMMAND DISAMBIGUATION MECHANISMS

Each of the four command disambiguation techniques we developed is based on a common conceptual back-end: For each operation where the system believes the user could commit a false feedforward error, a list of alternative operations is maintained. We call this list of alternative operations the *disambiguation record* for the operation.

In this paper we focus on exploring the design space of command disambiguation techniques, but there are multiple ways that the disambiguation records used by these techniques could be generated, ranging from simple manual methods to more sophisticated automatic techniques. Investigating these approaches is an interesting area for future work, and is discussed at the end of the paper.

The disambiguation techniques we introduce in this paper are *Detour Submenus* and *Detour Tooltips* (to assist the user before they select an incorrect command), *Dialog Link-Ups* and *Tool Swaps* (to help the user change course while using an incorrect command), and *Did-You-Mean History* + *Or-do* (to provide corrective action after performing an incorrect command). Each utilizes the disambiguation record to provide a displayed *disambiguation context* to assist with choosing the most appropriate command. We now present a detailed description of each technique.

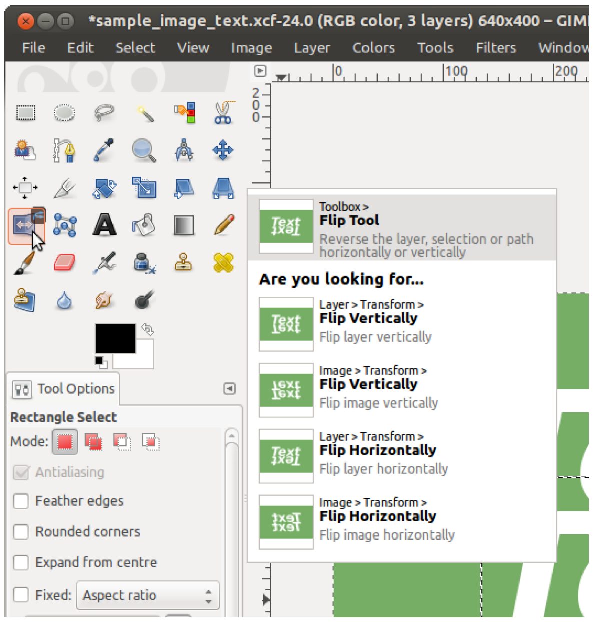
Detour Submenus and Tooltips

Detour Submenus embellish menu items by displaying the operation's disambiguation context (Figure 1, first page). The presence of alternatives is signaled by a small branching arrow icon at the far right of the menu item. Hovering the mouse over the menu item automatically displays the alternatives in a visually distinct extended menu, which users can use to directly invoke either the original command, or alternatives listed under an "Are you looking for…" heading.

When designing this mechanism, we considered a range of different ways to present a disambiguation context to the user before they invoked a command. For example, one alternative we considered was a text-based warning in the tooltip of the menu item. However, we ultimately pursued the current design because it provides a clear indicator that a menu item could be confused with something else, and enables the user to directly invoke alternatives.

In our prototype implementation, Detour Submenus appear immediately when hovering over a menu item, but a delay could be implemented to make the technique less aggressive. Schemes could also be explored for varying the delay based on the expertise and knowledge of the current user.

Detour Tooltips (Figure 5) use a similar approach to detour submenus, but are suitable for use on items in toolboxes, toolbars, and ribbon interfaces. When an item in the toolbox is hovered over for more than 300ms, a tooltip with its disambiguation context is displayed. The tooltip appears in a position where it does not obscure other items in the toolbox or toolbar (e.g., to its right), so the user is still free to explore other commands, a common practice that we have observed


*sample_image_text.xcf-24.0 (RGB color, 3 layers) 640x400 – GIM
File Edit Select View Image Layer Colors Tools Filters Window
Tool Options
Toolbox > Flip Tool
Are you looking for...
Layer > Transform > Flip Vertically
Image > Transform > Flip Vertically
Layer > Transform > Flip Horizontally
Image > Transform > Flip Horizontally

Reverse the layer, selection or path horizontally or vertically
Flip layer vertically
Flip image vertically
Flip layer horizontally
Flip image horizontally

Rectangle Select
Mode:
Antialiasing
Feather edges
Rounded corners
Expand from centre
Fixed: Aspect ratio

Figure 5. A *Detour Tooltip* for the Flip Tool in GIMP.

among participants engaged in trial-and-error problem-solving. When the cursor moves off of a command, the tooltip is closed after a delay of 1000ms, unless the cursor hovers over the tooltip in this time, in which case the tooltip persists so the user can interact with it (this is similar to the approach taken in other tooltip-based help systems [8, 24] and commercial applications such as Microsoft Office).

Dialog Link-Ups and Tool Hot-Swaps

For commands that require parameters through dialogs, *Dialog Link-Ups* augment dialog boxes with a disambiguation context that appears attached to the right side of the dialog box (Figure 6). Like Detour Submenus, the interface displays a list of alternative commands with previews of the result of each. When the user makes changes to parameters in the current dialog (e.g., adjusting *Width* in Figure 6 top), these are used to update the previews presented alongside the dialog. The idea is that these visual representations could help the user to recognize if the command they are using is inappropriate, and prompt them to choose an alternative command instead.

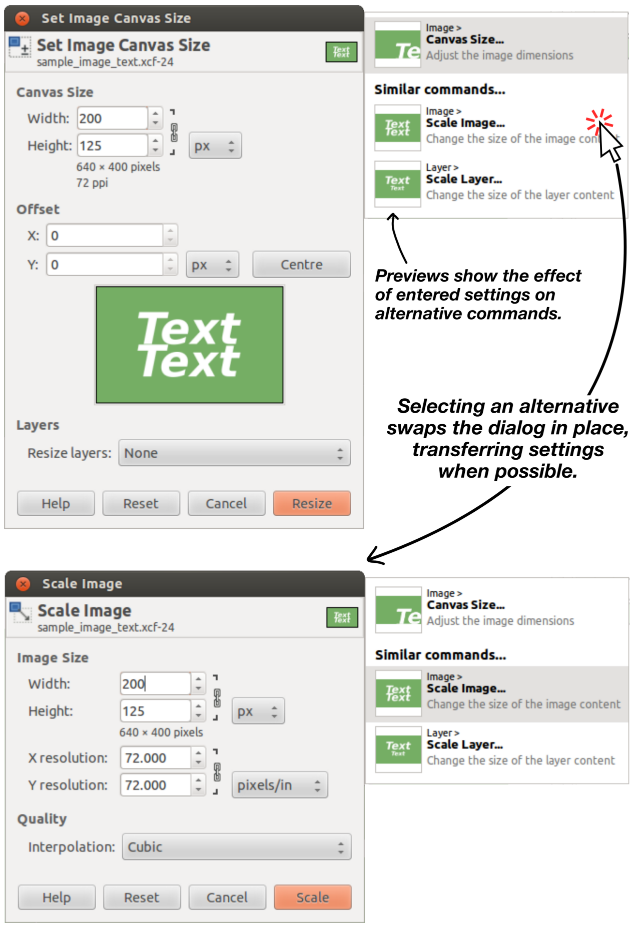
If the user clicks on an alternative command, the dialog box switches in-place to that of the chosen command, migrating parameters between like-labelled inputs from the two dialogs (e.g., when switching from Canvas Size to Scale Image, the *Width* and *Height* parameters are transferred).

Disambiguation contexts may not be symmetric between commands – two commands that could be confused with one another may have different disambiguation records. Thus, there is a question of whether the displayed disambiguation context should change when the user switches to a new dialog box using the dialog swapping mechanism. In our current design, we maintain the disambiguation context of the original command, for consistency.

Tool Hot-Swaps (Figure 7) provide similar functionality to Dialog Link-ups but for direct-manipulation tools. Once the user starts to apply a direct-manipulation tool to the document, a disambiguation context is displayed alongside the canvas. This allows the user to try out alternative tools to the one they are currently using. When an alternate tool is selected, clicks or strokes the user has already made are automatically migrated to the new tool. Figure 7 shows how this could be used with the Intelligent Scissors and Free Select tools in GIMP. A user partway through selecting a complex figure could realize that the Intelligent Scissors may not be the correct tool for the job, and switch to Free Select without needing to start their selection over from the beginning. Moreover, the user could evaluate the alternative tool and decide that their original choice was better, and return to it without re-performing any input.

Did-You-Mean History and Or-do

Our final technique, *Did-You-Mean History*, provides a way to retroactively switch to alternative commands after a command has been performed. When a command with a disambiguation record is executed, it is added to the Undo History list as usual, but with its alternatives listed under a “Did you mean…” heading (Figure 8). Selecting an alternative command in the Undo History list undoes the original



Adjust the image dimensions
Change the size of the image content
Change the size of the layer content

Canvas Size
Width: 200
Height: 125 px
640 × 400 pixels
72 ppi
Offset
X: 0
Y: 0 px Centre
Layers
Resize layers: None

Image Size
Width: 200
Height: 125 px
640 × 400 pixels
X resolution: 72.000
Y resolution: 72.000 pixels/in
Quality
Interpolation: Cubic

Previews show the effect of entered settings on alternative commands.

Selecting an alternative swaps the dialog in place, transferring settings when possible.

Figure 6. A *Dialog Link-Up* for the Set Image Canvas Size dialog in GIMP.

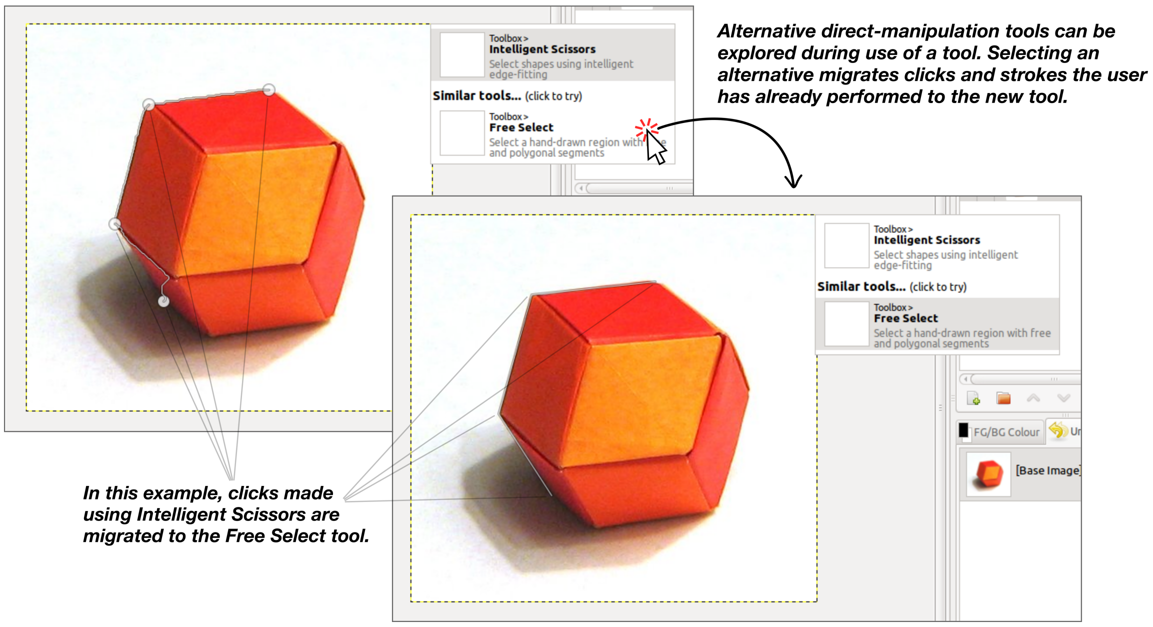

Alternative direct-manipulation tools can be explored during use of a tool. Selecting an alternative migrates clicks and strokes the user has already performed to the new tool.
In this example, clicks made using Intelligent Scissors are migrated to the Free Select tool.
Toolbox > Intelligent Scissors
Select shapes using intelligent edge-fitting
Similar tools... (click to try)
Toolbox > Free Select
Select a hand-drawn region with free and polygonal segments
FG/BG Colour
[Base Image]

Figure 5. ***Tool Hot-Swaps*** **allows the user to switch between direct-manipulation tools while they are being used, without re-performing input (such as re-selecting points around the object in this example).**

command, and substitutes the alternative in its place. An *Edit > Or-do* command can also be used to cycle through the set of alternatives in the disambiguation context. This command can also be invoked via a keyboard shortcut.

Command Disambiguation as a Holistic System

The interaction techniques that we have presented could support the user before, during, or after they have used an incorrect command. However, the techniques also naturally complement and support one another. For example, Detour Submenus can both serve as a way to access alternative commands in their own right, as well as a subtle cue that Or-do will have alternative options to cycle through after the command has been invoked.

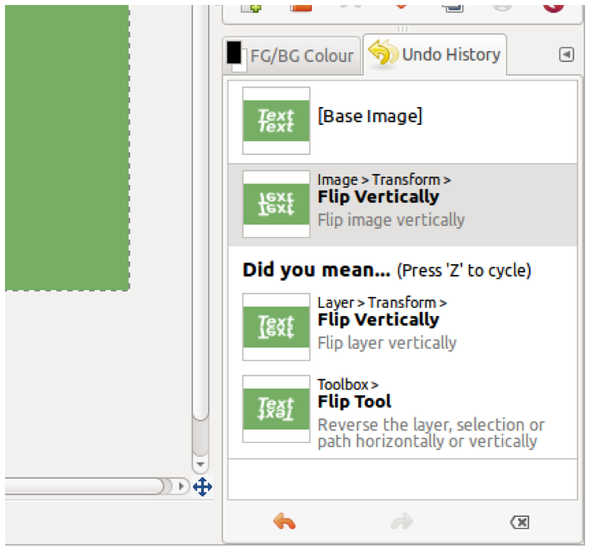


Figure 6. ***Did-You-Mean History*** **adds alternatives to the existing Undo History dialog, which can be selected or cycled through with the** ***Or-do*** **command.**

Prototype Implementation

Prototypes of the interaction techniques presented in this section were created in a mock GIMP interface that we developed for this project using HTML and JavaScript. We adopted this approach because our main objective was to investigate the space of designs for command disambiguation techniques. As such, speed of development and freedom from the constraints imposed by a particular interface toolkit were more important than demonstrating that the techniques could be applied in an actual application. Similar approaches have been used to investigate interaction techniques in the past (e.g., [3]). We note that, for the purposes of our study, our prototypes were very convincing. Many of our study participants attempted to interact with the mockup as if it was a fully functioning GIMP application, and one participant asked how we managed to recreate the GIMP interface in a web browser.

In the next section, we present the results of a qualitative user study we performed to gain feedback on our designs.

USER STUDY

To solicit reactions to the overall concept of command disambiguation techniques, and to gain feedback to refine our designs, we conducted a study with 12 participants (8 female, 4 male, ages 23–39) recruited through an on-campus mailing list targeted toward graduate students.

Each participant experienced three scenarios – flipping a layer, resizing an image, and selecting a complex figure – corresponding to the examples described in this paper and the video figure. For each scenario, we described the goal at a high level, then coached participants through interacting with the prototype. As they did so, they encountered the techniques we had developed, at which point we asked them to stop and answer questions on the perceived advantages and disadvantages of the techniques, in a semi-structured interview protocol, before continuing with the scenario. This approach enabled participants to experience first-hand what it would be like to interact with each of the techniques.

Results

Overall, there was wide agreement among participants that the techniques would be beneficial to users who are in the process of learning a software application. For example, P11 specifically stated that she would use the Detour Submenus if she came across them:

> *I like it, I think it's cool that it tells you where [the alternate command] is, so I would understand that it's under Layer, but I don't have to go there, I can just click it from here. I think that's really nice. I think, yeah it helps to avoid the confusion a lot, because I would definitely read these different options before I committed to selecting the first one.* – P11

Another participant mentioned that he would have benefitted from these techniques while learning Excel:

> *Thinking back to when I did learn to use Excel for the first time in a serious way, having that kind of functionality would have saved me a lot of time and hassle.* – P10

A number of specific features of the techniques were mentioned by participants. In particular, the preview images provided by the techniques were cited as a particularly important aspect of the system, as the following quotes demonstrate:

> *I think without the preview image I would still click 'Flip Vertically' as if... [pause] I would ignore it, I would be like 'ah, whatever, it's just giving me more text' right? So the fact that you have the previews here, I think is extremely helpful.* – P9

> *There is a lot of data here, like I mean text, in terms of instructions, like 'are you looking for X or Y'. And so when I'm looking at these two, I feel like [pause] it's good, but it requires me to think, and people probably won't necessarily always want to be thinking. [...] But I do notice that there are graphics to the left of them.* – P7

P10 cited the previews as being useful as well, and also expressed that, to a new user, text descriptions may be less useful than visual depictions when evaluating alternatives:

> *As I said, "Layer", "Image", to me, as a beginner user, those aren't terms with a great deal of meaning yet. But I know, for instance, if I was asked to make it look this way, when I open this up, that I can see, oh yeah, if I had done this, that wouldn't be what I was wanting, and if I had done that as well, but here's what I was looking for, so okay, that is called 'Layer > Transform > Flip Vertically'.* – P10

P10's comment suggests a related issue as well, that new users may have difficulty evaluating alternatives because they don't understand fundamental concepts that are used in the application, such as Layers and Images. This provides additional validation for our observation that incomplete conceptual understandings can be a cause of false-feedforward errors. It also raises an interesting design question: how could succinct visual (rather than text) depictions of fundamental concepts be developed for use in this kind of system.

Another feature appreciated by participants was the ability to switch between commands midway through use of a tool without re-entering information, as these quotes express:

> *This is great. I like this a lot. [...] I can click here [on an alternative], I can say 'oh yes, that's perfect, I'll stick with that', or alternatively, 'no, what I was doing before was what I wanted to be doing'. It's great that I don't have to go back and restart the task.* – P10

> *I like it, yeah. Because it gives me an alternative, and I can compare between the two tools to see which one is more powerful than the other. [...] In this scenario, I see that the Free Select is more powerful than the Intelligent Scissors then I will choose Free Select. It's just one click and then it solves the problem.* – P6

Given the popularity of this feature, it is worth investigating more flexible techniques for transferring input actions or settings between commands. For example, systems and toolkit-level techniques could be developed to enable settings migration between tools that achieve similar goals, but have a non-trivial mapping between their respective parameters.

Design Tensions

The most common concern raised by participants was that the techniques had the potential to add visual noise or clutter to the interface. This was typically mentioned as a concern of more experienced users of a system, as in this quote:

> *I mean, it's easy for the beginner, but, like, if I am experienced with Photoshop then I won't be confused with the different functions or different tools. So later I will not look at those extra choices, I will be right, I will be correct on my first try. [...] So then they will be annoyed by those extra tools.* – P6

However, this sentiment was not universal. P2, who self-identified as an experienced user, stated that she still falls victim to certain false feedforward errors, and felt the Detour Submenus could help her as well:

> *That was super useful because, I use GIMP a lot actually, I consider myself an expert, and I still go to Image > Flip Vertically instead of Layer > Flip Vertically all the time. So it was nice to have that little reminder.* – P2

Several participants indicated that the techniques could be problematic if they presented too many alternatives for a given command, or were applied on too many commands:

> *I think you're smart not to have too many [options] on there, maybe just the top three or four. Not even four, maybe just three that would be the most common ones people would want.* – P7

> *If it's on just one or two things that can be easily misinterpreted for one another, then that makes sense, but if it was on many things, I think that would actually complicate it more, because there would be so many options under every single command, that you'd be kind of overwhelmed, I think.* – P12

In future work, it would be worthwhile to investigate the tradeoffs between these two dimensions, to see when they provide benefit and when the benefit is outweighed by the added complexity the techniques create. The designs could also be refined to display only a few alternatives at a time, with a mechanism to access additional alternatives if needed.

Participants' comments also suggested a tension between limiting visual distraction and the need to catch the user's attention to indicate that alternatives exist. For example, while some participants mentioned that the techniques added visual noise, others overlooked them entirely, as in the following quote where P7 discusses Dialog Link-ups:

> *I like it, but I didn't at first notice it. I was so concentrated on this pop-up box [the dialog itself]. So I wasn't necessarily paying attention to what happened on the side.* – P7

A potential way to address this would be to give the techniques a visual style distinct from the rest of the interface (e.g., a distinct icon re-used across all techniques, or a color that does not occur frequently elsewhere in the interface), so that users could tune out these elements when they were not needed, but tune in to them when they are performing new or unfamiliar tasks.

A final concern raised by participants is that the shortcuts provided by command disambiguation techniques could become the standard way that users use a command, in place of its standard location in the interface:

> *But I think with this thing [the detour submenu], if there's a separate tool in the toolbox that does one of these [alternatives], and I discover this [detour submenu] first, then I'll probably just always end up doing this when I just want layer flip.* – P9

P11 mentioned this for the Or-do technique as well:

> *If I had this option, like maybe I would never actually learn to look here. And I wouldn't really learn much of the program. I would kind of just be used to, I would learn that it's always going to fix my mistake.* – P11

While this could happen, we do not believe it is a major concern – for infrequent users of the application, relying on shortcut methods may in fact be an economical strategy in terms of cognitive resources spent to accomplish a goal. We discuss this issue in greater detail in the next section.

DISCUSSION AND FUTURE WORK

The response to our prototype designs is encouraging – participants understood the intent of the techniques, expressed enthusiasm for them, and provided novel insights into their design. In this section we discuss some of these insights in greater detail, and outline areas for future work.

The observation by participants that they could fall into a pattern of using the shortcuts provided by command disambiguation mechanisms suggests an interesting tension in the interfaces of feature-rich software – the organization schemes used in these applications impose a logical and predictable ordering on the available commands, but these schemes do not serve all users equally. In particular, they best serve users that understand the fundamental underlying concepts in the application domain (e.g., what Layers are), and can be challenging to navigate for those who do not. Our techniques can be viewed as introducing a secondary, semantically meaningful organization of the application's functionality on top of its existing structure. An advantage of this two-level approach is that the secondary organization is free to do things that the original cannot – it can be redundant, presenting a given command in multiple places, or it can change to suit the individual user or situation, while the primary organization remains stable and predictable.

Given this two-level model, a key area for future work is to investigate how command disambiguation mechanisms could act as a platform for teaching fundamental concepts in the application domain. For example, the point at which the user has just committed (or almost committed) a false-feedforward error due to a conceptual misunderstanding is an ideal time to introduce fundamental domain knowledge to resolve this misunderstanding (e.g., by explaining the difference between Images and Layers). In this way, the techniques could help the user to transition to using the application's primary organization scheme over time.

A limitation of the techniques we have presented is that they only address recommending individual alternative commands for a given command. Scenarios could arise where a command falsely suggests a use that no one command can perform on its own, but instead requires the use of several commands together. For example, in GIMP there is no single command for drawing basic shapes such as rectangles or ellipses – one must first make a selection of the required shape, and then use additional commands to stroke or fill the selection. It would be interesting to explore how these one-to-many substitutions of commands could be conveyed. More generally, it would be interesting to quantify how often this situation occurs in popular feature-rich applications.

Finally, while we developed our prototypes in a mockup of the GIMP interface, we have ideas on how to address the key implementation details required for a fully working system.

First, methods will need to be developed to generate disambiguation records for a given application. At their simplest, observations of novice user behavior could be used to manually create a static set of disambiguation records. However, more sophisticated automated techniques could be explored as well, including: statically comparing command names, tooltip text, and icon images to identify similar commands; analyzing usage logs to identify sequences of actions where a user appears to have committed a false-feedforward error; and analyzing web-based instructional materials (such as tutorials) for warnings against confusing one tool with another. Techniques that model individual users' behavior could also be investigated, in order to create disambiguation records that are personalized to each user of an application.

Second, efficient methods for generating live previews will be required. This is particularly important because participants cited the previews as a key feature of our techniques. To achieve interactive speeds, the system could generate live previews in parallel background threads (as in [23]) or parallel instances of the application (as in [12]). If performance is still a bottleneck, “stock” previews, displaying representative examples of the effect of commands, could be displayed immediately, to allow a general comparison, then be replaced with live previews when ready.

To produce live previews for a range of different scenarios and types of commands (e.g., for direct manipulation tools, text tools, or non-visual tools), approaches could be adapted from past work on advanced previewing techniques [23, 24], text change visualization [4], and visualizing general changes in user interfaces [3].

CONCLUSIONS

False-feedforward errors are a serious problem for novice users of feature-rich software. In this paper, we have presented a set of command disambiguation techniques that explicitly address this type of error by making the user aware that alternative commands exist, and facilitating exploration of alternative commands before, during, and after execution of an incorrect command. The results of a qualitative study indicate that these techniques are considered valuable, and provided insights into key features and future directions for developing these techniques.

ACKNOWLEDGMENTS

We thank the reviewers of this work for their helpful suggestions. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

1. Akers, D., Jeffries, R., Simpson, M., and Winograd, T. Backtracking Events As Indicators of Usability Problems in Creation-Oriented Applications. *ACM Trans. Comput.-Hum. Interact. 19*, 2 (2012), 16:1–16:40.
2. Ames, A.L. Just what they need, just when they need it: an introduction to embedded assistance. *ACM SIGDOC ’01*, (2001), 111–115.
3. Baudisch, P., Tan, D., Collomb, M., et al. Phosphor: Explaining Transitions in the User Interface Using Afterglow Effects. *ACM UIST ’06*, (2006), 169–178.
4. Chevalier, F., Dragicevic, P., Bezerianos, A., and Fekete, J.-D. Using Text Animated Transitions to Support Navigation in Document Histories. *ACM CHI ’10*, (2010), 683–692.
5. Djajadiningrat, T., Overbeeke, K., and Wensveen, S. But How, Donald, Tell Us How?: On the Creation of Meaning in Interaction Design Through Feedforward and Inherent Feedback. *ACM DIS ’02*, (2002), 285–291.
6. Farkas, D.K. The role of balloon help. *ACM SIGDOC Asterisk J. Comput. Doc. 17*, 2 (1993), 3–19.
7. Gaver, W.W. Technology Affordances. *ACM CHI ’91*, (1991), 79–84.
8. Grossman, T. and Fitzmaurice, G. ToolClips: An investigation of contextual video assistance for functionality understanding. *ACM CHI ’10*, (2010), 1515–1524.
9. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S.R. Design As Exploration: Creating Interface Alternatives Through Parallel Authoring and Runtime Tuning. *ACM UIST ’08*, (2008), 91–100.
10. Kelleher, C. and Pausch, R. Stencils-based tutorials: Design and evaluation. *ACM CHI ’05*, (2005), 541–550.
11. Lafreniere, B., Bunt, A., and Terry, M. Task-centric interfaces for feature-rich software. *ACM OZCHI ’14*, (2014), 49–58.
12. Laput, G., Adar, E., Dontcheva, M., and Li, W. Tutorial-based interfaces for cloud-enabled applications. *ACM UIST ’12*, (2012), 113–122.
13. Linton, F., Joy, D., Schaefer, H.-P., and Charron, A. OWL: A recommender system for organization-wide learning. *Educational Technology & Society 3*, 1, 62–76.
14. Lunzer, A. and Hornbæk, K. Subjunctive Interfaces: Extending Applications to Support Parallel Setup, Viewing and Control of Alternative Scenarios. *ACM Trans. Comput.-Hum. Interact. 14*, 4 (2008), 17:1–17:44.
15. Mankoff, J., Hudson, S.E., and Abowd, G.D. Interaction Techniques for Ambiguity Resolution in Recognition-based Interfaces. *ACM UIST ’00*, (2000), 11–20.
16. Mankoff, J., Hudson, S.E., and Abowd, G.D. Providing Integrated Toolkit-level Support for Ambiguity in Recognition-based Interfaces. *ACM CHI ’00*, 368–375.
17. Marks, J., Andalman, B., Beardsley, P.A., et al. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. *ACM Press/Addison-Wesley SIGGRAPH ’97*, (1997), 389–400.
18. Matejka, J., Li, W., Grossman, T., and Fitzmaurice, G. CommunityCommands: Command recommendations for software applications. *ACM UIST ’09*, (2009), 193–202.
19. Nancel, M. and Cockburn, A. Causality: A Conceptual Model of Interaction History. *ACM CHI ’14*, (2014), 1777–1786.
20. Norman, D.A. *The Design of Everyday Things.* Basic Books, New York, 2002.
21. Novick, D.G., Andrade, O.D., and Bean, N. The microstructure of use of help. *ACM SIGDOC ’09*, 97–104.
22. Schwarz, J., Hudson, S., Mankoff, J., and Wilson, A.D. A Framework for Robust and Flexible Handling of Inputs with Uncertainty. *ACM UIST ’10*, (2010), 47–56.
23. Terry, M. and Mynatt, E.D. Side Views: Persistent, On-demand Previews for Open-ended Tasks. *ACM UIST ’02*, (2002), 71–80.
24. Terry, M., Mynatt, E.D., Nakakoji, K., and Yamamoto, Y. Variation in Element and Action: Supporting Simultaneous Development of Alternative Solutions. *ACM CHI ’04*, (2004), 711–718.
25. Vermeulen, J., Luyten, K., van den Hoven, E., and Coninx, K. Crossing the Bridge over Norman’s Gulf of Execution: Revealing Feedforward’s True Identity. *ACM CHI ’13*, (2013), 1931–1940.