Open Archive Toulouse Archive Ouverte

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is an author's version published in:
http://oatao.univ-toulouse.fr/24921

Official URL

DOI : https://doi.org/10.1145/3331162

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr

Fortunettes: Feedforward about the Future State of GUI Widgets

SVEN COPPERS, Hasselt University - tUL - Flanders Make, Expertise Centre for Digital Media

KRIS LUYTEN, Hasselt University - tUL - Flanders Make, Expertise Centre for Digital Media

DAVY VANACKEN, Hasselt University - tUL - Flanders Make, Expertise Centre for Digital Media

DAVID NAVARRE, University of Toulouse, ICS - IRIT

PHILIPPE PALANQUE, University of Toulouse, ICS - IRIT and Eindhoven University of Technology, Department of Industrial Design
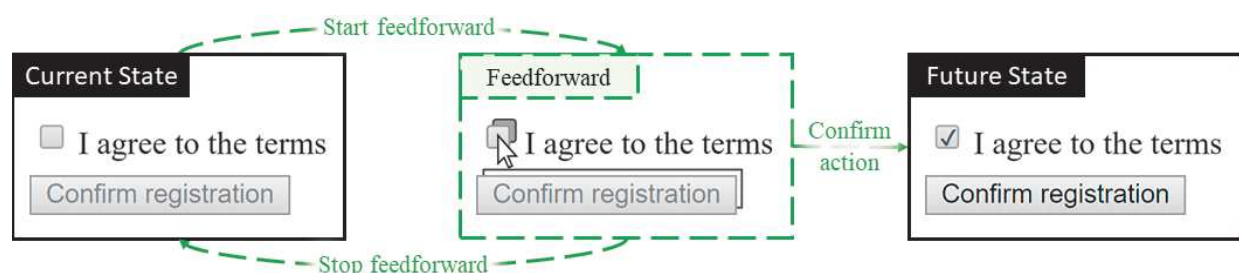
CHRISTINE GRIS, Airbus Operations SAS

Fig. 1. Feedforward at the widget level to provide an answer to the *what if* question "What will happen if I click this checkbox?".

Feedback is commonly used to explain *what happened* in an interface. *What if* questions, on the other hand, remain mostly unanswered. In this paper, we present the concept of enhanced widgets capable of visualizing their future state, which helps users to understand what will happen without committing to an action. We describe two approaches to extend GUI toolkits to support widget-level feedforward, and illustrate the usefulness of widget-level feedforward in a standardized interface to control the weather radar in commercial aircraft. In our evaluation, we found that users require less clicks to achieve tasks and are more confident about their actions when feedforward information was available. These findings suggest that widget-level feedforward is highly suitable in applications the user is unfamiliar with, or when high confidence is desirable.

Keywords: Feedforward, Intelligibility, User Interface Widgets

Authors' addresses: Sven Coppers, Hasselt University - tUL - Flanders Make, Expertise Centre for Digital Media, Diepenbeek, Belgium, sven.coppers@uhasselt.be; Kris Luyten, Hasselt University - tUL - Flanders Make, Expertise Centre for Digital Media, Diepenbeek, Belgium, kris.luyten@uhasselt.be; Davy Vanacken, Hasselt University - tUL - Flanders Make, Expertise Centre for Digital Media, Diepenbeek, Belgium, davy.vanacken@uhasselt.be; David Navarre, University of Toulouse, ICS - IRIT, Toulouse, France, navarre@irit.fr; Philippe Palanque, University of Toulouse, ICS - IRIT, Toulouse, France, Eindhoven University of Technology, Department of Industrial Design, Eindhoven, The Netherlands, palanque@irit.fr; Christine Gris, Airbus Operations SAS, Toulouse, France, christine.gris@airbus.com.

1 INTRODUCTION

Considering their ever-growing variety and complexity, applications can be hard for users to understand. Complexity, however, should not be avoided, but rather tamed with well-designed user interfaces [42]. One key aspect of UI design is feedback, as it informs the user about the outcome of an action after it has been completed [55]. Feedback essentially provides an answer to *what happened* questions and has been identified as a critical step to cross the Gulf of Evaluation in Norman's Stages of Action model [43]. Feedback is widely supported by GUI toolkits, since they provide means to display both inherent feedback (e.g. the new application state that results from an user action, such as a window disappearing from the screen when closing it), and explicit feedback (e.g. error messages, such as "Username already exists" or "Invalid password").

Providing answers to *what if* questions is less prevalent. Undo, redo and history mechanisms can be used to explore what the result of an action will be [41], but such strategies are cumbersome and time consuming [23]. A more convenient approach is the use of feedforward, which shows what the result of an action will be *before* that action is performed [18]. Feedforward helps to cross Norman's Gulf of Execution [55] and enables the user to explore, understand and directly assess alternatives [27], which results in improved confidence and awareness. This is particularly useful for novices, since they can be characterized by little planning and relying on the order in which actions come to mind [23, 48]. Despite the clear benefits, feedforward is usually limited to cognitive affordances such as labels and tooltips. In some cases, custom feedforward is provided such as previewing markup options for rich-text in Microsoft Office applications, or direct manipulation tasks such as drag and drop. Rich, informative feedforward, however, is not nearly as established as feedback and support for it is largely absent in GUI toolkits.

In this paper, we present an approach to provide rich widget-level feedforward about the future state of widgets. Figure 1 shows a small example of this kind of widget-level feedforward: when the user hovers the checkbox, the button visualizes its future state in terms of availability (the button will become enabled when the user clicks), while the checkbox visualizes its future state in terms of value (the checkbox will become selected when the user clicks). With this information, the user knows that to enable the 'Confirm registration' button, the 'I agree to the terms' box needs to be checked first. Thus, widget-level feedforward provide an answer to *what if* questions like "What will happen if I click this checkbox?", by presenting what the result of the user action will be before that action is performed. In this paper, we put forward the following contributions:

(1) the concept of enhanced GUI widgets capable of visualizing their future state;
(2) a set of exemplary widgets, called 'Fortunettes', capable of visualizing their future state;
(3) a description of two complimentary approaches to provide toolkit-level support for such feedforward;
(4) examples that illustrate the usefulness of Fortunettes in applications with varying levels of complexity;
(5) a comparative evaluation between Fortunettes and identical widgets without a feedforward layer.

2 RELATED WORK

In this section, we highlight the importance of intelligibility, describe earlier research regarding relevant mechanisms such as feedback and feedforward, and provide an overview of approaches to generate feedforward.

2.1 Intelligibility, Feedback and Feedforward

Having a limited understanding of an application's behavior can be frustrating [6, 28], and causes users to lose trust [3, 36]. Making matters worse, previous work points out that users are reluctant to devote time exclusively to learn and understand an application, and prefer to learn on-the-fly instead [13, 14]. Intelligibility and scrutability, however, can improve the understanding of an application by providing in-situ explanations [1, 5, 11, 31]. Lim et al. describe different types of explanations, including *why, what*, and *what if* explanations [30]. They also put forward a set of guidelines for explanations, such as providing explanations automatically and supporting combinations of explanations [30, 45]. Most applications already provide some kind of feedback to explain *what* the result of an action is after that action has been completed. In cases where users fail to notice, feedback mechanisms such as Phosphor [10] can be used to add afterglow effects on top of UI widgets to provide feedback that fades over time.

What if explanations are less common, with users mainly using undo, redo and history mechanisms to explore what the result of an action will be [41]. If users find results unexpected, insufficient or wrong, they simply revert to the previous state. This strategy is usually cumbersome and time consuming. Providing feedforward to inform the user about the result of an action before the action becomes final [18, 51] can drastically decrease the need for such strategies, and at the same time support in-situ learning by answering *what if* questions [30]. The design space of feedforward resembles the design space of feedback [17], and the information they present can be very similar.

2.2 How to Provide Feedforward

Although equally important as feedback, feedforward is rarely explicitly supported by UI toolkits [55]. As a result, feedforward is usually either very basic, limited to certain WYSIWYG elements or special purpose. Basic feedforward is already present in interfaces in the form of labels and icons, such as "open in new tab" ⤢. However, they convey a limited amount of information and ignore the dynamic needs of the user. Hence, more informative and interactive feedforward is desirable [27]. Simple hover effects and tooltips are often used for this purpose. More comprehensive examples include Balloon help [20], ToolClips [24], and Stencils [25], which greatly increase the amount of information that can be conveyed.

Several graphical interfaces already include custom designed feedforward. In the Microsoft Office package, for example, some of the markup changes can be previewed through hovering a markup option without activating it. Side Views [53] are enhanced tooltips that move beyond previewing a single future state by visualizing multiple future states at the same time. Side Views provide previews for one action or multiple actions chained together, and when hovering over a set of parameters shows what effect the parameter will have on the resulting state. However, Side Views do not scale well when feedforward is needed about multiple objects that might be affected by an action.

Some applications embed special-purpose feedforward tailored to their specific use cases. In gestural interfaces, for example, the gestures are often hard to discover and learn. To accommodate in-situ learning, OctoPocus [9] and Gestu-Wan [49] dynamically visualize the possible paths a user can follow to complete specific gestures. Similarly, TouchGhosts [54] facilitate discovering

and learning multi-touch interactions by visualization not only the necessary actions, but also the effects of those actions on the current system state.

Complementary to the above-mentioned forms of feedforward, there is great potential for adding feedforward about the future state at the level of GUI widgets, since these widgets are the primary channel for communicating the application logic.

2.3 Generating Feedforward

To provide feedforward that is representative of an action's actual outcome, a model of the application logic is required. A large body of literature on model-based design already exists, including notations such as task models [33, 34], finite state machines [56], Petri nets [8], interface description languages [32], and user manual markup languages [37]. For ad-hoc implementations, a formal behavioral model can be reconstructed using external tools that observe and analyze the interface behavior [12, 26, 52]. Not only are there many ways to model application logic, but also many paradigms to implement this logic, which is infamously difficult [38]. The most common approach is the event-callback system, which is widely adopted by toolkits. Unfortunately, this tends to produce error-prone spaghetti code, because the implementation is spread across many locations [35, 40].

With many ways to model and implement application logic, it can be challenging to integrate and align the model and implementation with one another. Interaction designers and programmers often think in terms of states [29, 39, 50] instead of modules. With ConstraintJS [46] and InterState [47], Oney et al. provide tools to model application logic in terms of states, and UI events are mapped onto transitions between those states. SwingStates add support for Finite State Machines (FSMs) to the widely used Java Swing Toolkit [4]. All existing approaches use the single state principle, requiring an application to be in exactly one state at any point in time [53].

3 THE CONCEPT OF WIDGET-LEVEL FEEDFORWARD

In this section, we illustrate the concept of widgets that are capable of presenting their future state as feedforward, and we present the interaction pattern that is required to use them.

3.1 Proof of Concept Widget-level Feedforward Visualization

To provide answers to *what if* questions about an action that is under consideration, widgets need to be able to present their future state in addition to their current state. We considered a number of designs, such as copying the entire widget or integrating the cues inside the widget. As a proof of concept, we decided to condense the feedforward information into a *feedforward layer* stacked behind the widget itself, although other ways of integrating and visualizing feedforward are definitely possible.

Figure 2 presents the anatomy of our proof of concept visualization. The border of the feedforward layer expresses *future availability* (Figure 2a): a dashed gray line indicates that the widget will become disabled in the future, while a full black line indicates the widget will become enabled. The inside region of the feedforward layer presents the *future value* (Figure 2b). For a checkbox or radio button, for instance, a white background indicates that it will not be selected in the future, whereas a darker background indicates that it will be selected. This kind of feedforward about selections is particularly useful for listboxes, as they typically support multiple selection through modifier keys (e.g. '*CTRL*' or '*SHIFT*'). The feedforward changes dynamically when a modifier key is pressed, thereby clarifying the effect it will have on the future selection. The future selection can be shown in the feedforward layer for all options in the listbox, including options that are currently outside the viewport controlled by the scrollbar.

The proof of concept visualization is not without limitations. It is only conveys information about future availability and value, and feedforward can only be perceived when the corresponding widget
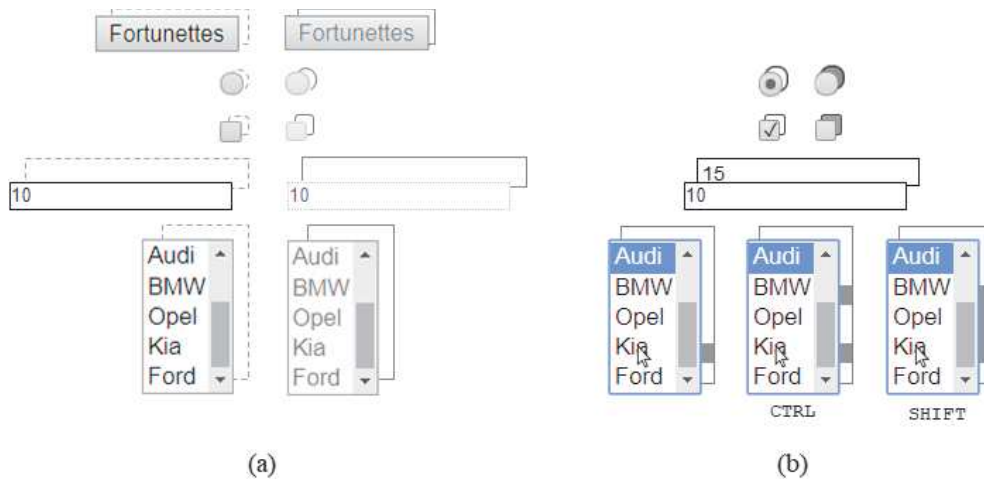
Fig. 2. Widgets capable of providing feedforward about their future state in terms of (a) *availability* and (b) *value*. Availability is presented through the markup of the border of the feedforward layer: a dashed gray border indicates that the widget will become disabled, a full black border that the widget will become enabled. The future value is shown in the inside region of the feedforward layer, for instance through the background color.

itself is visible. These limitations can be addressed by more elaborate or alternative visualizations, which are beyond the focus of this paper.

3.2 Interacting with Widget-level Feedforward

When users perform an action in a GUI without feedforward, such as clicking on a widget, the interface simply transitions from the current state to the future state (Figure 3a). The new state is only presented to the users after the action has been performed. When including feedforward in the GUI, the feedforward information does not need to be presented permanently, but should be triggered by the user on a *by-need* basis [17], when users show an intention to interact with a part of the GUI. To facilitate such widget-level feedforward, a mediation process is needed that between intention to perform an action (i.e. action samples) and confirming that action (i.e. final actions) [51]. In our approach, this process requires an intermediary feedforward phase, as well as three intermediary events (Figure 3b): (1) start showing feedforward when the user is considering to perform an action, (2) stop showing feedforward when the user is not longer considering that action, or (3) confirm and actually execute the considered action.

(a) Without feedforward
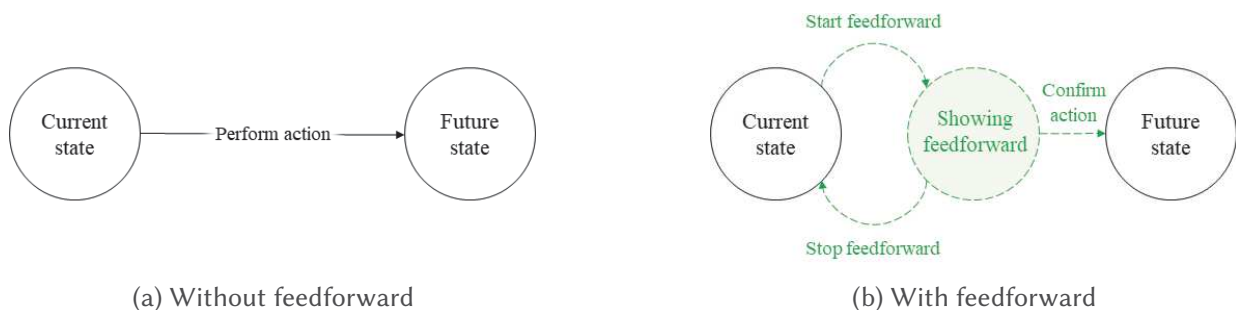
(b) With feedforward

Fig. 3. (a) A traditional interaction model in which the next state is triggered immediately. (b) Widget-level feedforward presents a glimpse of the future without committing to an action, by inserting an intermediary feedforward phase and three intermediary events that detect the intention to interact.

When and how widget-level feedforward should be triggered is an important decision, since these triggers need to be transient and reversible [51]. Detecting when a user starts to consider an action can be achieved in various ways: hovering over a widget with the mouse cursor, a long press on a touch screen, or gazing at a widget when eye tracking is available, for example. Detecting when a user is no longer considering an action can be achieved in the opposite manner: when the mouse cursor leaves the widget, when the widget looses focus, or when the user is no longer gazing at the widget.

Widget-level feedforward is not limited to the widget that is acted upon by the user, but covers all widgets that are affected by that action (e.g. in the example of Figure 1, both the checkbox and the button provide feedforward). To reduce '*noise*' and make changes detectable at a glance, only affected widgets are enhanced with a feedforward layer. The absence of a feedforward layer thus communicates that the state of a particular widget will not change. This behavior can be customized, for instance only showing feedforward for actions that are less likely to be understood by a user, or using a separate trigger to activate feedforward (e.g. long hover or holding '*CTRL*' on the keyboard).

4 PROVIDING TOOLKIT SUPPORT FOR FEEDFORWARD

To support widget-level feedforward for an action, three intermediary events need to be handled, as described in Section 3.2: start showing feedforward, stop showing feedforward, and confirming the action. In this section, we describe two approaches to extend GUI toolkits to support these events: automated toolkit-driven feedforward and implementing feedforward manually using the toolkit. We demonstrate both approaches using the example presented in Figure 4: when the user is logged in, a message can be written in the textbox or the user can log out. If the textbox is not empty, the message can be sent. Sending the message clears the textbox. Figure 4b presents the dialog model [22] of this example, including preconditions between square brackets, similar to the semantics of Augmented Transition Networks [56].
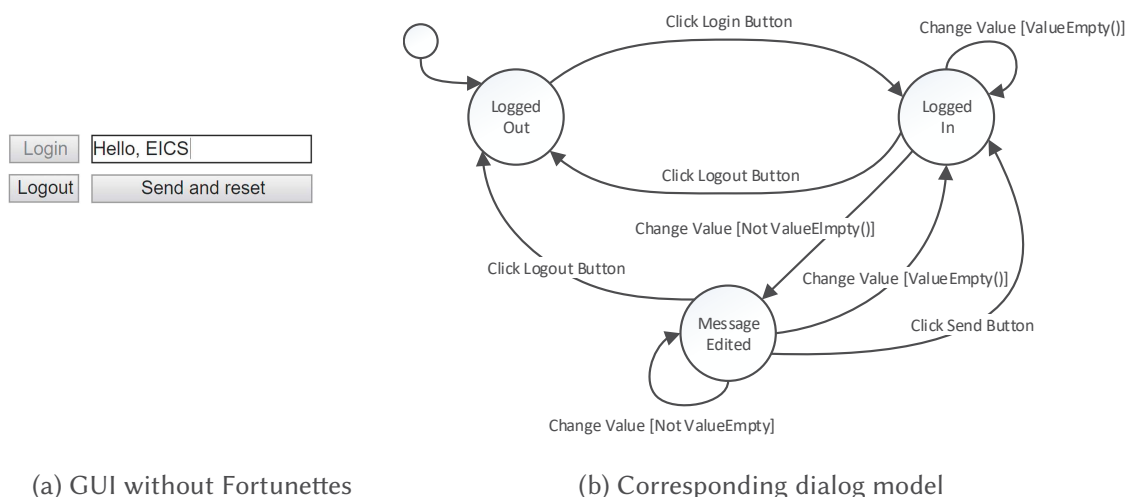

(a) GUI without Fortunettes   (b) Corresponding dialog model

Fig. 4. In this example without feedforward, typing something in the textbox will immediately enable the "Send and reset" button.

4.1 Toolkit-driven Feedforward

Feedforward for an action can only be presented to the user if the outcome of that action is known beforehand. The implementation of this dialog is usually already encapsulated in event handlers, which are typically used by GUI toolkits to map user commands to system actions. These event handlers might invoke more formal models, such as Petri nets [7] or Finite State Machines [44]. In our toolkit-driven approach, we reuse existing event handlers to deduce the outcome of an action. To this end, the event that is used as a trigger to start feedforward (e.g. hovering the 'Send and reset' button) can also be used as an *alias* for performing the action that is under consideration (e.g. clicking the 'Send and reset' button). In other words, to deduce what feedforward to show when hovering the 'Send and reset' button, the event handler that handles a click on the 'Send and reset' button is called to determine the future state.

To achieve this, both events are sent to the toolkit, along with the desired mode ('feedforward' or 'execute'), as depicted in Figure 5. The toolkit routes both types of events to the same event handler in the application, which returns the future state of each affected widget to the toolkit. This is an important prerequisite for our approach: the event handlers need to return those future states, instead of actually modifying the widgets themselves. Next, depending on the mode, the toolkit either presents that state as feedforward (e.g. the textbox will become empty) or sets it as the new state (e.g. the textbox is emptied) by modifying the corresponding properties of the widgets.
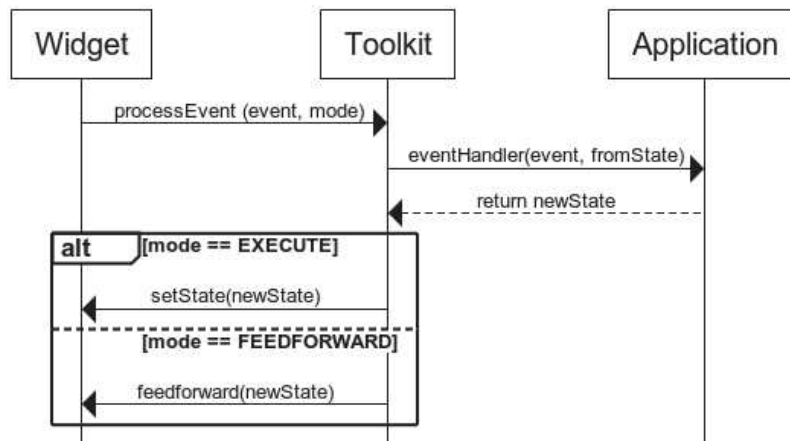

Fig. 5. When event handlers are used for both handling actions and providing feedforward about those actions, the toolkit decides whether their outcome is set as the new state or shown as feedforward.

Figure 6a presents a screenshot of the interface of our example when the application is in the 'Message Edited' state. When hovering the 'Send and Reset' button, the toolkit predicts a transition to the 'Logged In' state, and therefore provides feedforward about the textbox becoming empty and the 'Send and Reset' button becoming disabled. Figure 6b shows how the dialog model is extended by the toolkit to support feedforward. The original states (solid black circles) are no longer connected through events directly, but through intermediary feedforward phases and events (dashed green circles and arrows).
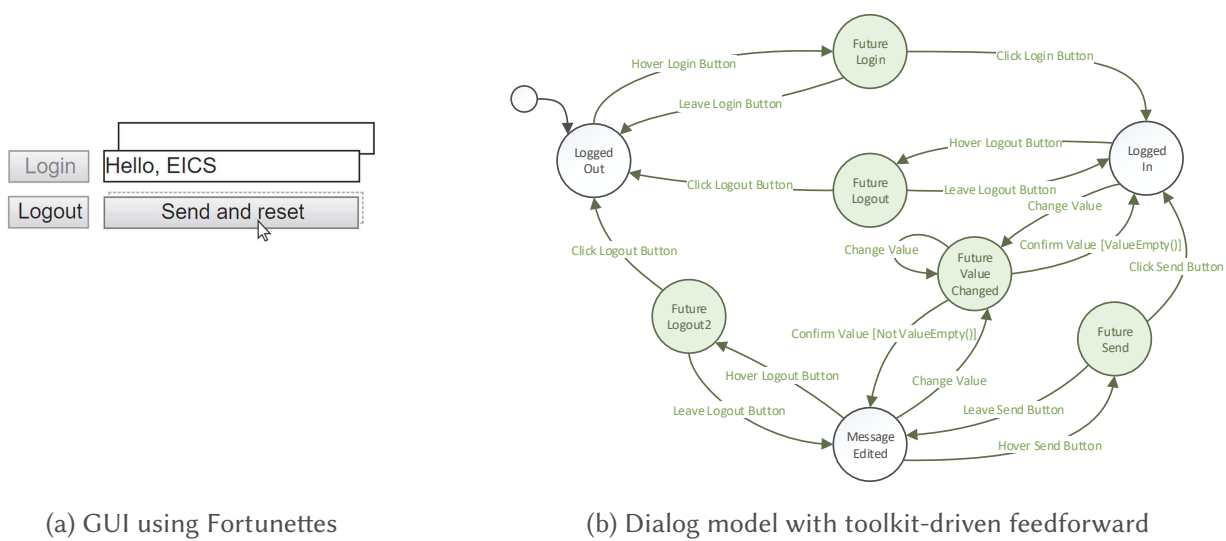
(a) GUI using Fortunettes   (b) Dialog model with toolkit-driven feedforward

Fig. 6. (a) In this screenshot of the example, feedforward shows that clicking the 'Send and reset' button will empty the textbox and disable the 'Send and reset' button. (b) The toolkit automatically inserts an intermediary feedforward phase and three intermediary events for every action in the original dialog model, without any effort from the developer.

4.2 Specifying Feedforward Manually

Toolkit-driven feedforward avoids that developers have to implement all feedforward manually and ensures that feedforward is representative for the actual outcome of an action. It cannot be used, however, when feedforward is undesirable or when custom feedforward is required. In addition, toolkit-driven feedforward is unsuitable for event handlers that invoke irreversible behavior. In such cases, custom feedforward handlers such as `startFeedforwardWidgetClicked()`, `stopFeedforwardWidgetClicked()`, and `confirmWidgetClicked()` can be specified to emulate the behavior of their event handler counterparts or to provide custom feedforward.
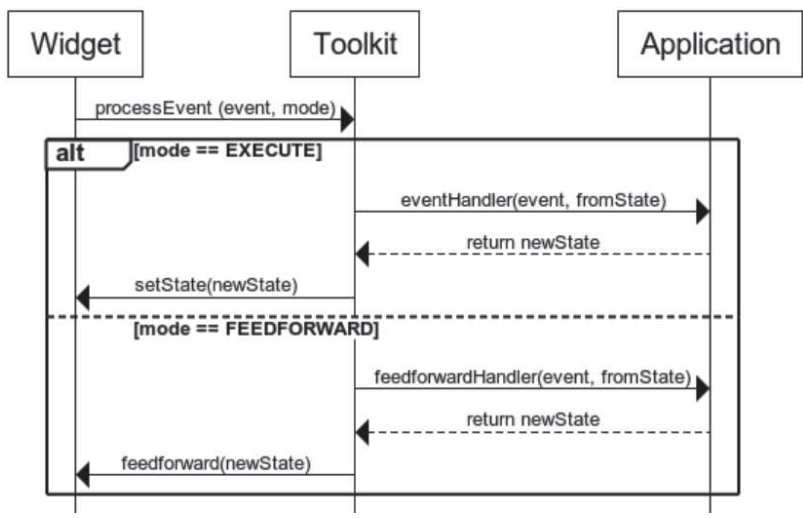

Fig. 7. When the developer writes custom feedforward handlers, both the current state and next state of a widget can be modified.

An example of irreversible behavior can be found in Figure 6: hovering 'Send and reset' would actually send the message to its destination, because the event handler that handles a click on that button is used to determine the future state for the purpose of feedforward. To avoid this, the developer can provide a custom feedforward handler that emulates the existing event handler, without actually sending the message. These custom feedforward handlers override the toolkit-driven feedforward and are shown in orange in Figure 8 and Figure 9. In contrast to our example, which is focused on the UI behavior, real-world examples are likely to have more irreversible event handlers, and will thus require developers to specify more feedforward handlers based on existing code.
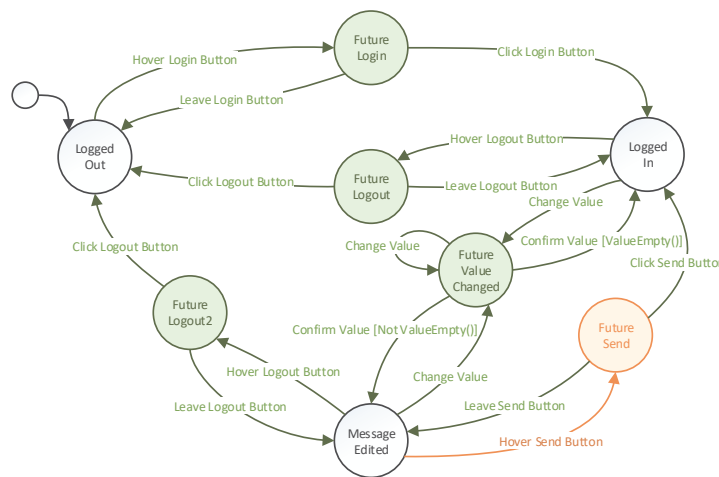

Fig. 8. Automated toolkit-driven feedforward (green dashed states and transitions) can be overridden by the developer by implementing custom feedforward handlers (orange dashed transition).

```
1    class LoginController extends Controller {
2        initState(newWindowState) { … }
3
4        loginButtonClicked(event, newWindowState) { … }
5        logoutButtonClicked(event, newWindowState) { … }
6        messageValueChanged (event, newWindowState) { … }
7
8        sendButtonClicked (event, newWindowState) {
9            this.sendMessage(value); // Actually send the message
10
11           newWindowState.getWidgetState("message").value = "";
12           newWindowState.getWidgetState("send_and_reset").enabled = false;
13       }
14
15       previewSendButtonClicked (event, newWindowState) {
16           newWindowState.getWidgetState("message").futureValue = "";
17           newWindowState.getWidgetState("send_and_reset").enabled = false;
18       }
19   }
```

Fig. 9. An overview of all event and preview handlers to implement the login example. Green event handlers are existing event handlers that can be reused for feedforward. The developer only needs to implement the orange feedforward handler, because the black event handler irreversibly calls sendMessage() and can therefore not be reused by the toolkit.

4.3 Proof of Concept Implementations

We implemented both aforementioned approaches to extend two different types of GUI toolkits: the Java Swing library, and HTML + JavaScript. Figure 10 depicts a button in various states in both toolkits. The extended version of Java Swing was used to develop the example of the weather radar (Section 5), while the web toolkit was used for various examples of our online evaluation (Section 6).

Fig. 10. An example of feedforward for buttons in each of our implemented toolkits.

Both toolkits use the Decorator design pattern [21] to add feedforward rendering capabilities to the widgets. The basic idea is to *decorate* the original widget with an additional feedforward layer within a single container, as shown in Figure 10. Similar to accessors to modify the state, such as setEnabled(boolean x) and setValue(String s)), we added accessors such as setFeedforwardEnabled(boolean x) and setFeedforwardValue(String s) to modify the state of the feedforward layer, which can be used by both the toolkit (as described in Section 4.1) and the developer (as described in Section 4.2). This approach is reflected in the domain model in Figure 11, as the WidgetView needs to be able to visualize the combination of exactly two instances of WidgetState: the instance that represents the current state, and the instance that represents the future state.

Fig. 11. Each WidgetView is capable of visualizing two instances of WidgetState simultaneously: the current and the future state.

Some widgets can be decomposed in '*subwidgets*' that are closely related. For instance, when selecting a radio button in a radio group, all other radio buttons in that group will be deselected. Thus, for these type of widgets WidgetBehavior needs to be able to manipulate its own WidgetState directly for feedforward purposes as well (Figure 11).

5 THE COCKPIT WEATHER RADAR CASE

To demonstrate the usefulness of widget-level feedforward in more complicated applications, we used our toolkits to replicate a standardized interface to control the weather radar in commercial aircraft.

5.1 Context and Use

Cockpits of commercial aircraft are equipped with a wide variety of interfaces to support pilots during their flight. One of these interfaces is the weather radar, which is used to increase awareness of meteorological phenomena, for example to avoid storms or heavy precipitations. Figure 12a presents the interface to control it, as well as the output it produces (colored forms that show the position, thickness and size of the clouds ahead of the aircraft). The control interface complies with the ARINC 661 standard [2] that specifies, among others, widget properties and communication

protocols for cockpit applications. The dialog [8, 22] of such applications is complex, as it merges the pilots' needs (to perform their mission) and the behavior of aircraft systems (that is controlled by these applications).

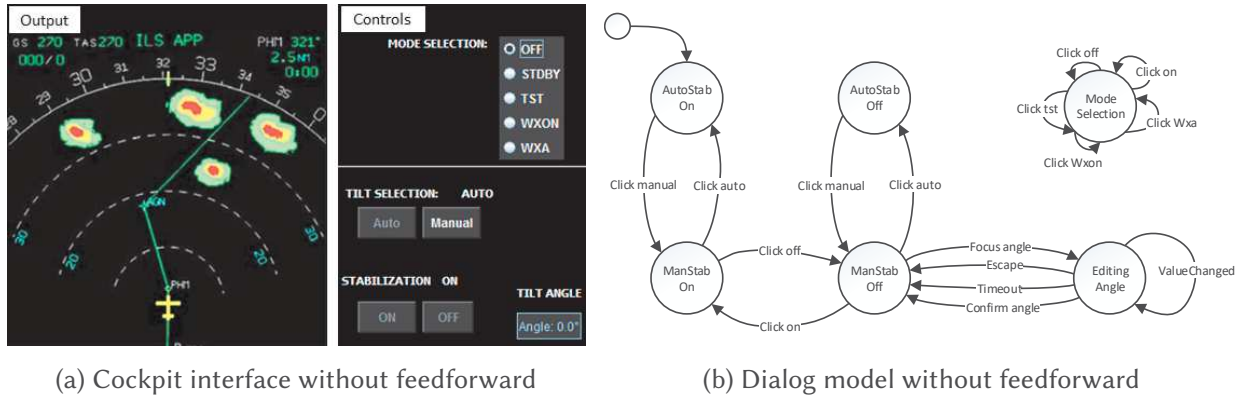(a) Cockpit interface without feedforward   (b) Dialog model without feedforward

Fig. 12. (a) The output and controls of an existing cockpit weather radar interface used in commercial aircraft. (b) Dialog model describing the behavior of the cockpit weather radar interface.

The control interface (right-hand side of Figure 12a) provides two functions to the crew. The first one is the mode selection of the weather radar: the five radio buttons at the top of the pane show the current status of the radar and can be used to switch from one mode to another. The second function, in the lower part of the pane, is dedicated to the orientation (or 'tilt angle') of the physical radar. The tilt angle needs to be adjusted when the aircraft is changing altitude, as the pilot might want to check the weather condition at the targeted altitude. Weather radars, however, have physical constraints that (usually) do not allow them to vary more than 15 degrees from the aligned position. For this reason, the application '*clamps*' the tilt angle between +15 and −15 degrees. If the crew sets an angle larger than +15 or smaller than −15 degrees, the upper or lower bound will be selected instead. To set a tilt angle, the tilt selection must be in manual mode and stabilization, which aims at keeping the radar beam stable even in case of turbulence, must be off.

The buttons in the lower part of the window influence each other, which is modelled in the dialog model in Figure 12b. The initial state corresponds to the user interface in Figure 12a. In that state (AutoStabOn), the only available action is to click on the 'Manual' button, which disables the autonomous behavior of the weather radar and sets it to manual mode (ManStabOn). Next, clicking on the 'OFF' button disables stabilization (ManStabOff), making it possible to enter a value for the tilt angle. The corresponding textbox employs the principle of '*caging*', which means that all other widgets are disabled while editing its value. The new tilt angle is confirmed by pressing ENTER, or discarded after a timeout.

5.2   Adding Feedforward to the Weather Radar

To illustrate the use of widget-level feedforward, we replicated the weather radar interface using Fortunettes. Consider the three steps in Figure 13. We start in the 'ModeSelection' state, with the 'OFF' mode selected. (Step 1) When hovering the 'WXA' radio button, the feedforward layers reveal that 'WXA' mode will become selected and 'OFF' will become deselected, based on the default behavior of a radio button group. In addition, the lack of other feedforward layers indicates that no other widgets are affected by the mode change, in accordance to the application model. Steps 2 and 3 demonstrate behavioral dependencies in the dialog of this application and how Fortunettes can support users in identifying them. (Step 2) While hovering the 'MANUAL' button, the feedforward

layers show that the stabilization 'OFF' button will become available and that tilt selection can be reverted to automatic, as the 'AUTO' button will become available. (Step 3) After clicking on the 'MANUAL' button, the 'OFF' button becomes available. Hovering 'OFF' shows that the textbox for the tilt angle will become available.
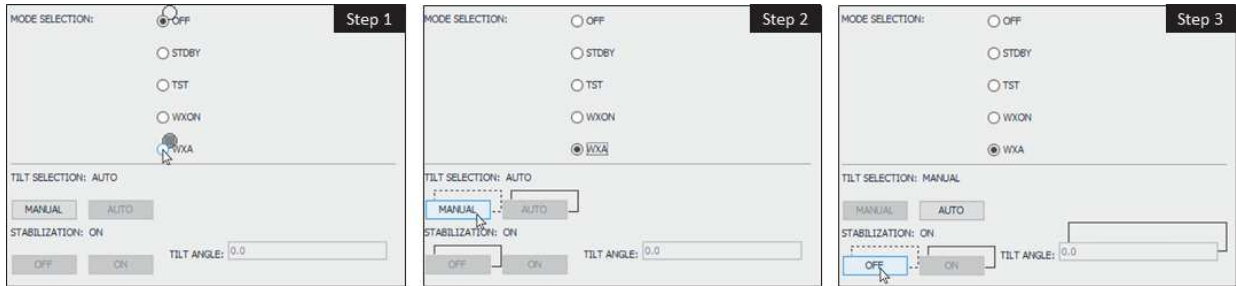

Fig. 13. A replicated version of the cockpit weather radar using Fortunettes. (Step 1) Hovering the 'WXA' radio button shows that changing mode does not impact widgets outside of the radio button group. (Step 2) Hovering the 'MANUAL' button reveals that the stabilization 'OFF' button will become available. (Step 3) Hovering 'OFF' reveals that the tilt value will become editable.
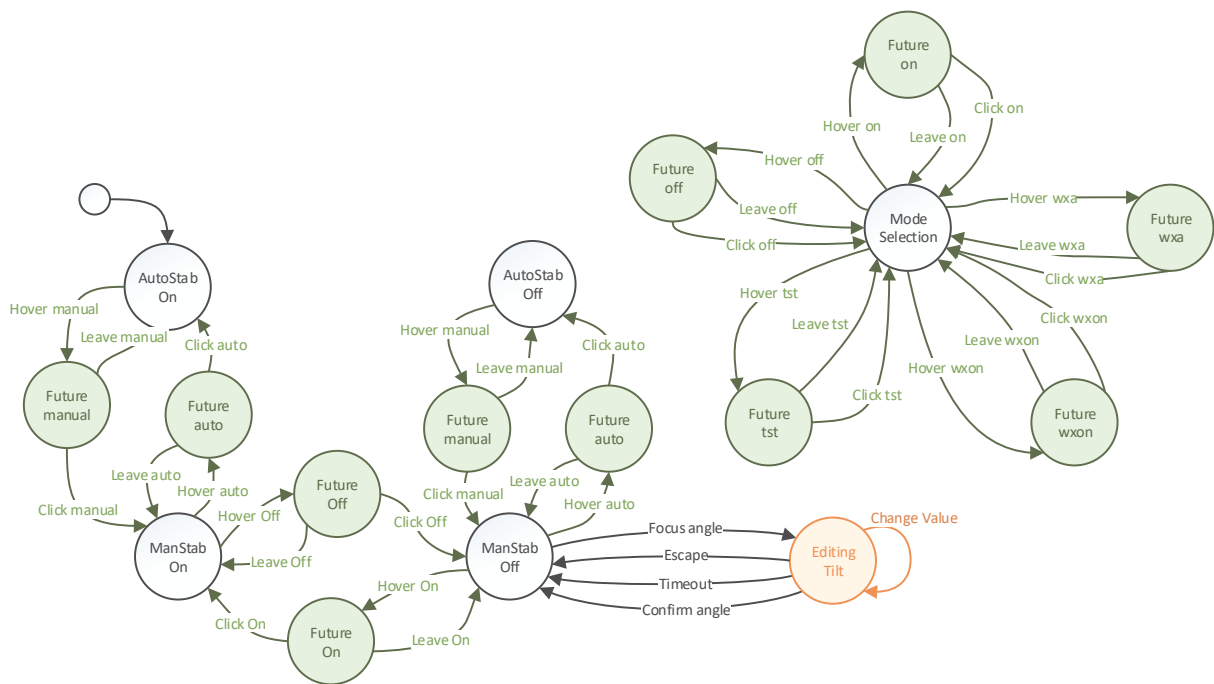

Fig. 14. Dialog model describing the behavior of the cockpit weather radar interface extended with feedforward. The green dashed states and transitions are toolkit-driven and do not require any developer effort. The orange dashed state and transition represent custom feedforward, which is implemented manually by the developer.

Following the pattern presented in Section 3, the addition of feedforward results in the dialog model in Figure 14. The left-hand side represents the extended behavior of the lower part of the pane, while the right-hand side represents the extended behavior of the mode selection at the top of the pane. We only have to specify one custom feedforward handler (shown in orange in Figure 14

```
1    class WXRController extends Controller {
2        initState(newWindowState) { … }
3
4        modeOffSelected(event, newWindowState) { … }
5        modeStdbySelected(event, newWindowState) { … }
6        modeTstSelected(event, newWindowState) { … }
7        modeWxonSelected(event, newWindowState) { … }
8        modeWxaSelected(event, newWindowState) { … }
9        tiltManualClicked(event, newWindowState) { … }
10       tiltAutoClicked(event, newWindowState) { … }
11       stabilizationOnClicked(event, newWindowState) { … }
12       stabilizationOffClicked(event, newWindowState) { … }
13
14       // Caging behavior
15       tiltAngleFocussed(event, newWindowState) { … }
16       tiltAngleEscaped(event, newWindowState) { … }
17       tiltAngleTimeout(event, newWindowState) { … }
18
19       tiltValueChanged(event, newWinowState) {
20           int tiltAngle = min(15, max(-15, newWinowState.getWidgetState("tilt").value));
21           newWinowState.getWidgetState("tilt").value = tiltAngle;
22
23           this.physicalRadar.setTiltAngle(tiltAngle); // Start physical motion
24       }
25
26       previewTiltValueChanged(event, newWinowState) {
27           int tiltAngle = min(15, max(-15, newWinowState.getWidgetState("tilt").value));
28           newWinowState.getWidgetState("tilt").value = tiltAngle;
29       }
30   }
```

Fig. 15. An overview of all event and preview handlers to implement the cockpit weather radar. Green event handlers are existing event handlers that can be reused for feedforward. The developer only needs to implement the orange feedforward handler, because the black event handler is irreversible and cannot be reused by the toolkit.

and Figure 15) for manipulating tilt, because reusing the existing event handler would actually manipulate the physical part of the weather radar. As a result of adding feedforward, pilots get a preview of the new tilt angle, which might be clamped between +15 and −15, before confirming the tilt angle.

As the example shows, the feedforward information increases awareness, which in turn can contribute to the overall safety [19]. Implementing this feedforward requires little effort from the developer: only a single custom feedforward handler needs to be implemented, whereas all other feedforward is toolkit-driven by reusing existing event handlers.

6   EVALUATION

We performed a comparative study to evaluate the impact of Fortunettes on the user experience. The study contained a wide range of demo applications to explore the limits of our approach and to find out what kind of applications benefit the most from feedforward.

6.1   Participants

We recruited 104 participants via social media and mailing lists. 10 did not participate in all steps of the experiment and were therefore discarded. Participants were randomly assigned to two groups for our mixed-design experiment. The first group had 16 female and 33 male participants (5 aged 18-20; 29 aged 21-29; 8 aged 30-39; 3 aged 40-49; 4 aged 50-59), with varying backgrounds (e.g. students, researchers, education, health, administration and government). The second group had 15 female and 30 male participants (1 aged 18-20; 30 aged 21-29; 9 aged 30-39; 1 aged 40-49; 4 aged

50-59), with varying backgrounds as well (e.g. students, researchers, engineering, business, art, logistics). All participants had experience with using computers to browse the web, handle e-mails and/or create documents.

6.2 Procedure

We opted for an online survey for higher response rates and convenience. To alleviate the effects of participating in an uncontrolled environment [15], we recruited a large number of participants and performed standard outlier removal[1]. Moreover, we limited the number of tasks to keep the experiment short (17 minutes on average) and ensure a low drop-out rate [16].

The procedure of the survey is outlined in Figure 16. Participants were first briefed about the study's purpose and our data policy, based on the GDPR legislation[2]. After giving their informed consent, participants completed a short demographic survey (*DEM*) and a three-minute tutorial (*TUT*) on how to interpret widget-level feedforward. Both groups of participants then performed a short task (*TSK* in Table 1) in each of the five different interfaces shown in Figure 17. The tasks had varying levels of difficulty in terms of minimum number of clicks required and familiarity with the domain. For *quantitative* between-subject comparison, the first group of participants received widget-level feedforward during these tasks, whereas the second group used ordinary widgets without feedforward. After completing the tasks, both groups switched conditions and performed new tasks (*TSK′* in Table 1) that required (partial) re-exploration of the interface. Due to possible learning effects, this second set of tasks is not used for any quantitative comparisons, but is only intended to allow for *qualitative remarks* in a closing survey that compares the two sets of widgets (*COM*).
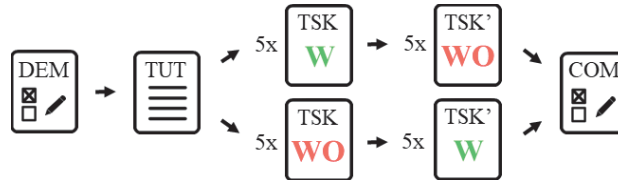

Fig. 16. The online survey started with a demographic survey (*DEM*) and tutorial (*TUT*). Each group performed five short tasks in one condition (*TSK*), followed by five tasks in the opposite condition (*TSK′*). The study ended with a comparative survey (*COM*).

6.3 Measurements and Analysis

Participants were explicitly instructed to click as few times as possible to achieve each task. We expect that the number of clicks will decrease when users have a better understanding of the impact of their interactions. Without knowledge of the domain or feedforward, participants need trial-and-error, and will thus click more often to find the solution. Therefore, the number of clicks becomes an indicator of user understanding. Besides logging the actual number of clicks and hovers, we also logged the time to completion of the task. After each task, participants filled in a short survey about their perceived performance, the perceived difficulty, understanding of the application, confidence, and their strategy to find the solution to the task.

In our statistical analysis, we only consider results of the (*TSK*) phase (i.e. only the first time a participant encounters an application) to eliminate training effects. Since half of the participants

---

[1]yatani.jp/teaching/doku.php?id=hcistats:outlierdetection

[2]https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en

received feedforward, and the other half did not, we looked for between-subject differences. We used for Chi-square and Mann-Whitney U tests to determine the significance of those differences, since Shapiro tests found our data does not follow a normal distribution. We removed outliers from each condition when their number of clicks or completion time differed more than two times the standard deviation from the mean of the other participants in the same condition. On average, 4.5 participants ($< 10\%$) were removed from each condition (group $\times$ task).
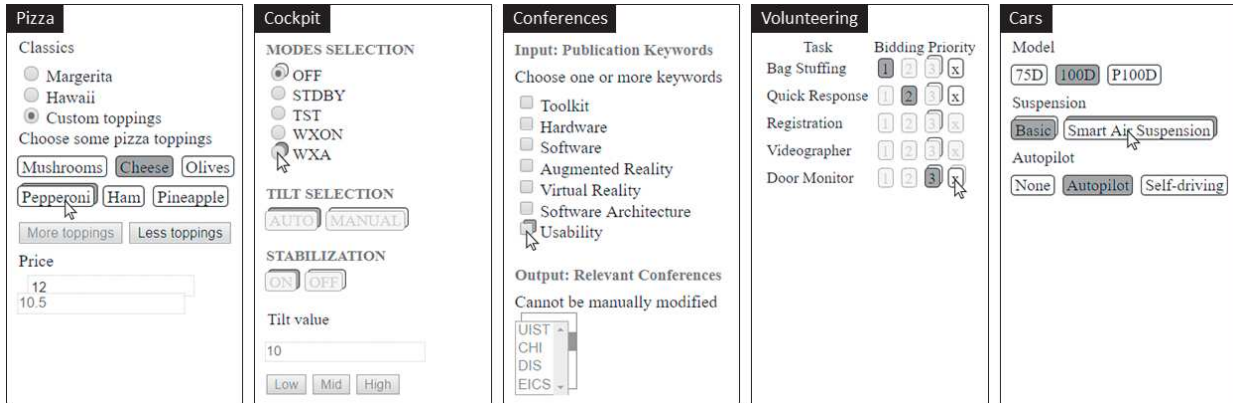

Fig. 17. Screenshots of the five use cases in which each participant performed a task with and without the feedforward.

	TSK	TSK'
Pizza	Create pizza with exactly *3 toppings* and *price = 12*	Create pizza with exactly *3 toppings* and *price = 14,5*
Cockpit	Set tilt value to *15*	Set tilt value to *25* [with different application behavior]
Conferences	Select keywords so *CHI and EICS* are the suggested conferences	Select keywords so *UIST and DIS* are the suggested conferences
Volunteering	Set bidding priority for *Videographer at 3*	Set bidding priority for *Registration at 2 and 3*
Cars	Configure car with *self-driving capabilities and basic suspension*	Configure car with *smart air suspension and autopilot*

Table 1. For each of the tasks in our experiment, participants were asked to click as few times as possible.

6.4 Results and Discussion

Table 2 and Figure 18 provide an overview of the between-subject comparison. Participants with the feedforward layer clicked significantly less often to achieve their tasks compared to participants without ($U = 13368, Z = -7.37, p < 0.001, r = 1.90$). Completion times, on the other hand, tend to increase in the condition with the feedforward layer. Only for the 'Cars' interface, the increase is significant ($U = 1220.5, Z = 3.58, p < 0.001, r = 0.92$). An increase is to be expected, since participants were asked to perform tasks with the least amount of clicks possible (i.e. low error rate) instead of in the shortest time (i.e. high speed of performance). 6 participants confirmed that they required more time to process the additional feedforward information, while P21 ($WO{\rightarrow}W$) "*wanted to discover all the possibilities [...] when the second layer was present*". Regardless of their measured performance, 71.3% of all participants perceived that the feedforward layer helped them to achieve tasks more quickly.

The measured performance aligns well with the self-reported strategies: significantly more participants used a "*trial-and-error*" strategy ($\chi^2(1, N = 446) = 66.30, p < 0.001, \phi = 0.38$) when the feedforward layer was not available (70.7%) compared to when it was available (32.6%). P10 ($WO{\rightarrow}W$), P18 ($W{\rightarrow}WO$) and P26 ($WO{\rightarrow}W$) complained that the impact of their choices was unclear, while P2 ($W{\rightarrow}WO$) stated: "*I used mostly trial-and-error if I didn't know what the buttons*

	Average Number of Clicks				Average Time (s)		
	Optimum	Without	With	p-value	Without	With	p-value
Pizza	3	9.9	**7.0**	**< 0.005**	47.5	64.0	> 0.05
Cockpit	3	7.1	**4.2**	**< 0.001**	31.6	35.7	> 0.05
Conferences	2	9.2	**2.6**	**< 0.001**	35.3	39.9	> 0.05
Volunteering	2	3.1	**2.3**	**< 0.001**	24.8	30.3	> 0.05
Cars	2	3.3	3.7	> 0.05	**22.3**	33.9	**< 0.001**
Global (avg)	2.4	6.5	**4.0**	**< 0.001**	**32.2**	41.0	**< 0.001**

Table 2. Between-subject comparison of performance between conditions with (W) and without (WO) feedforward layer in terms of the number of clicks and completion time. Only tasks in the TSK phase are considered, to eliminate possible training effects. The 'Optimum' represents the minimum number of clicks required to solve the task.

were about to do". When feedforward was available, however, participants relied significantly less on their own interpretation of the application domain ($U = 18807, Z = -4.15, p < 0.001, r = 1.01$) and more on the feedforward itself. It helped them in finding the solution (P31, $W{\to}WO$) and made them more confident (P79,$W{\to}WO$; P75,$W{\to}WO$). Although we did not measure any significant difference in perceived difficulty (Figure 18), these findings suggests that widget-level feedforward is highly suitable for applications that are unfamiliar to the user or when high confidence is desirable. In the closing survey (COM), 79.8% of the participants confirmed that widget-level feedforward would be useful for unfamiliar software. In contrast, only 17.0% believe widget-level feedforward would be useful for software they already know well. When asked in what kind of applications they would (not) want to have the feedforward layer, multiple participants stated that they would like to have feedforward in forms with constraints on their input, such as online tax declarations or configuration interfaces for cell phone subscriptions.
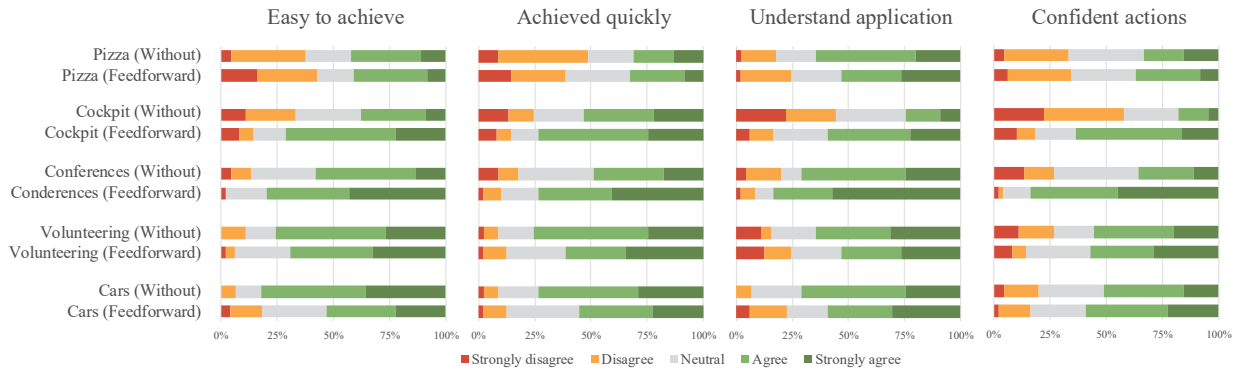

Fig. 18. After completing each task, participants responded to Likert-scale questions about the perceived ease of use, performance, understanding of the application and confidence.

7 LIMITATIONS AND FUTURE WORK

The current implementation and proof-of-concept visualization of Fortunettes have a few constraints that present interesting opportunities for future work.

Feedforward about a single action. In our concept, feedforward information only reveals the outcome of the next action that the user is considering, whereas multiple actions can be required to achieve a task. More advanced mechanisms that guide users through these actions, such as widget-level feedforward about consecutive actions or automatically generating step-by-step instructions, are compelling directions of future work.

A limited number of properties is visualized. Our proof-of-concept visualization is currently limited to visualizing the immediate future in terms of availability and value of a widget. The visualization could support other widget properties that might be useful to the user as well, such as position, visibility, or the appearance of a new window/tab/dialog.

Widgets must be visible. Feedforward information about the future state of a widget cannot be perceived by the user when the widget itself is invisible (e.g. due to the current scroll position, or because it is positioned inside a different tab or window). In some cases, a more elaborate visualization in the visible parent container widget could include aggregated feedforward information (e.g. the tab bar can show some feedforward information about what is inside the other tab pages). In other cases, new approaches to explore feedforward information are needed.

8 CONCLUSION

In this paper, we introduce the concept of GUI widgets capable of providing feedforward about their own future state, based on an event that might happen in the interface. To enable designers and developers of interactive systems to integrate feedforward, we describe a proof-of-concept visualization and two complimentary approaches to provide toolkit support. The first approach is to allow the GUI toolkit to generate feedforward automatically, by reusing existing event handlers. While this approach saves developers from implementing all feedforward manually, it is inadequate when feedforward is irrelevant, when custom feedforward is needed, or when the event handlers are irreversible and cause side effects. The second approach allows refinement of the behavior by specifying custom feedforward handlers that emulate the behavior of existing event handlers. We implemented two feedforward toolkits that apply these approaches: one toolkit extends Java Swing, whereas the other adds feedforward capabilities in HTML + JavaScript. To demonstrate the applicability of these toolkits in more complicated applications, we successfully replicated an interface that controls the weather radar in commercial aircraft.

A comparative evaluation with 94 participants shows that Fortunettes lead to less clicks to achieve specified tasks. As predicted by the trade-off between error rate and speed of performance, the completion times were negatively affected, in contrast to the increased perceived performance. After 3 minutes of training, 80.8% of the participants relied on feedforward when confronted with unfamiliar applications. Feedforward makes GUIs more predictable and improves the confidence users have in their actions, especially when confronted with new interfaces. 79.8% of the participants confirmed that Fortunettes would be useful for unfamiliar software.

Our results highlight the contrast between feedback and feedforward: while feedback provides the necessary confirmation about the execution of an action, feedforward is particularly useful to increase confidence before executing the action. Feedforward is thus particularly important when the cost of mistakes is high (e.g. in terms of money, safety, or health). Our findings suggest that widget-level feedforward is highly suitable in applications the user is unfamiliar with, or when high confidence is desirable.

REFERENCES

[1] Ashraf Abdul, Jo Vermeulen, Danding Wang, Brian Y. Lim, and Mohan Kankanhalli. 2018. Trends and Trajectories for Explainable, Accountable and Intelligible Systems: An HCI Research Agenda. In *Proceedings of the 2018 CHI*

Conference on Human Factors in Computing Systems (CHI '18). ACM, New York, NY, USA, Article 582, 18 pages. https://doi.org/10.1145/3173574.3174156

[2] Incorporated (ARINC) Aeronautical Radio. 2016. *ARINC661 Cockpit Display System Interfaces to User Systems ARINC Specification 661, supplement 6*. Standard. AEEC - Engineering Standards for Aircraft Systems.

[3] Stavros Antifakos, Nicky Kern, Bernt Schiele, and Adrian Schwaninger. 2005. Towards Improving Trust in Context-aware Systems by Displaying System Confidence. In *Proceedings of the 7th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '05)*. ACM, New York, NY, USA, 9–14. https://doi.org/10.1145/1085777.1085780

[4] Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: Adding State Machines to the Swing Toolkit. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 319–322. https://doi.org/10.1145/1166253.1166302

[5] Mark Assad, David J. Carmichael, Judy Kay, and Bob Kummerfeld. 2007. PersonisAD: Distributed, Active, Scrutable Model Framework for Context-Aware Services. In *Pervasive Computing*, Anthony LaMarca, Marc Langheinrich, and Khai N. Truong (Eds.). Vol. 4480. Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72. http://link.springer.com/10.1007/978-3-540-72037-9_4

[6] Louise Barkhuus and Anind Dey. 2003. Is Context-Aware Computing Taking Control away from the User? Three Levels of Interactivity Examined. In *UbiComp 2003: Ubiquitous Computing: 5th International Conference, Seattle, WA, USA, October 12-15, 2003. Proceedings*, Anind K. Dey, Albrecht Schmidt, and Joseph F. McCarthy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–156. https://doi.org/10.1007/978-3-540-39653-6_12

[7] Rémi Bastide and Philippe A. Palanque. 1990. Petri Net Objects for the Design, Validation and Prototyping of User-driven Interfaces. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction (INTERACT '90)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 625–631. http://dl.acm.org/citation.cfm?id=647402.725296

[8] Rémi Bastide and Philippe A. Palanque. 1996. Implementation Techniques for Petri Net Based Specifications of Human-Computer Dialogues. In *Computer-Aided Design of User Interfaces I*, Jean Vanderdonckt (Ed.). 285–302.

[9] Olivier Bau and Wendy E. Mackay. 2008. OctoPocus: A Dynamic Guide for Learning Gesture-based Command Sets. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1449715.1449724

[10] Patrick Baudisch, Desney Tan, Maxime Collomb, Dan Robbins, Ken Hinckley, Ken Hinckley, Maneesh Agrawala, Shengdong Zhao, and Gonzalo Ramos. 2006. Phosphor: Explaining Transitions in the User Interface Using Afterglow Effects. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1166253.1166280

[11] Victoria Bellotti and Keith Edwards. 2001. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human-Computer Interaction* 16, 2 (Dec. 2001), 193–212. https://doi.org/10.1207/S15327051HCI16234_05

[12] Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology (UIST '15)*. ACM, New York, NY, USA, 259–268. https://doi.org/10.1145/2807442.2807473

[13] John M. Carroll and Mary Beth Rosson. 1987. *Paradox of the Active User*.

[14] Richard Catrambone and John M. Carroll. 1987. Learning a Word Processing System with Training Wheels and Guided Exploration. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface (CHI '87)*. ACM, New York, NY, USA, 169–174. https://doi.org/10.1145/29933.275625

[15] Scott Clifford and Jennifer Jerit. 2014. Is There a Cost to Convenience? An Experimental Comparison of Data Quality in Laboratory and Online Studies. *Journal of Experimental Political Science* 1, 2 (2014), 120–131. https://doi.org/10.1017/xps.2014.5

[16] Frédéric Dandurand, Thomas R. Shultz, and Kristine H. Onishi. 2008. Comparing online and lab methods in a problem-solving experiment. *Behavior Research Methods* 40, 2 (01 May 2008), 428–434. https://doi.org/10.3758/BRM.40.2.428

[17] William Delamare, Céline Coutrix, and Laurence Nigay. 2015. Designing Guiding Systems for Gesture-based Interaction. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 44–53. https://doi.org/10.1145/2774225.2774847

[18] Tom Djajadiningrat, Kees Overbeeke, and Stephan Wensveen. 2002. But How, Donald, Tell Us How?: On the Creation of Meaning in Interaction Design Through Feedforward and Inherent Feedback. In *Proceedings of the 4th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS '02)*. ACM, New York, NY, USA, 285–291. https://doi.org/10.1145/778712.778752

[19] Mica R. Endsley. 2011. *Designing for Situation Awareness: An Approach to User-Centered Design, Second Edition* (2nd ed.). CRC Press, Inc., Boca Raton, FL, USA.

[20] David K. Farkas. 1993. The role of balloon help. *ACM SIGDOC Asterisk Journal of Computer Documentation* 17, 2 (May 1993), 3–19. https://doi.org/10.1145/154425.154426

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[22] Mark Green. 1986. A Survey of Three Dialogue Models. *ACM Trans. Graph.* 5, 3 (July 1986), 244–275. https://doi.org/10.1145/24054.24057

[23] T. R. G. Green. 1989. Cognitive Dimensions of Notations. In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*. Cambridge University Press, New York, NY, USA, 443–460. http://dl.acm.org/citation.cfm?id=92968.93015

[24] Tovi Grossman and George Fitzmaurice. 2010. ToolClips: An Investigation of Contextual Video Assistance for Functionality Understanding. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1515–1524. https://doi.org/10.1145/1753326.1753552

[25] Caitlin Kelleher and Randy Pausch. 2005. Stencils-based Tutorials: Design and Evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*. ACM, New York, NY, USA, 541–550. https://doi.org/10.1145/1054972.1055047

[26] Andrew J. Ko and Brad A. Myers. 2010. Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Methodology* 20, 2 (Aug. 2010), 1–36. https://doi.org/10.1145/1824760.1824761

[27] Benjamin Lafreniere, Parmit K. Chilana, Adam Fourney, and Michael A. Terry. 2015. These Aren't the Commands You're Looking For: Addressing False Feedforward in Feature-Rich Software. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and; Technology (UIST '15)*. ACM, New York, NY, USA, 619–628. https://doi.org/10.1145/2807442.2807482

[28] Jonathan Lazar, Adam Jones, and Ben Shneiderman. 2006. Workplace user frustration with computers: an exploratory investigation of the causes and severity. *Behaviour & Information Technology* 25, 3 (May 2006), 239–251. https://doi.org/10.1080/01449290500196963

[29] Catherine Letondal, Stéphane Chatty, Greg Philips, Fabien André, and Stéphane Conversy. 2010. Usability requirements for interaction-oriented development tools. In *PPIG 2010, 22nd Annual Workshop on the Psychology of Programming Interest Group*. 12–16.

[30] Brian Y. Lim and Anind K. Dey. 2010. Toolkit to Support Intelligibility in Context-aware Applications. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp '10)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/1864349.1864353

[31] Brian Y. Lim, Anind K. Dey, and Daniel Avrahami. 2009. Why and Why Not Explanations Improve the Intelligibility of Context-aware Intelligent Systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 2119–2128. https://doi.org/10.1145/1518701.1519023

[32] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Murielle Florins. 2004. USIXML: A User Interface Description Language Supporting Multiple Levels of Independence.. In *ICWE Workshops*. 325–338.

[33] Kris Luyten, Tim Clerckx, Karin Coninx, and Jean Vanderdonckt. 2003. Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. In *Interactive Systems. Design, Specification, and Verification*, Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–217.

[34] Célia Martinie, David Navarre, Philippe Palanque, and Camille Fayollas. 2015. A Generic Tool-supported Framework for Coupling Task Models and Interactive Applications. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 244–253. https://doi.org/10.1145/2774225.2774845

[35] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. *ACM SIGPLAN Notices* 44, 10 (Oct. 2009), 1. https://doi.org/10.1145/1639949.1640091

[36] Bonnie M. Muir. 1994. Trust in Automation: Part I. Theoretical Issues in the Study of Trust and Human Intervention in Automated Systems. *Ergonomics* 37, 11 (Nov. 1994), 1905–1922. https://doi.org/10.1080/00140139408964957

[37] Lars Müller, Ilhan Aslan, and Lucas Krüßen. 2013. GuideMe: A Mobile Augmented Reality System to Display User Manuals for Home Appliances. In *Advances in Computer Entertainment*. Vol. 8253. Springer International Publishing, Cham, 152–167. https://doi.org/10.1007/978-3-319-03161-3_11

[38] Brad Myers, Scott E. Hudson, Randy Pausch, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. https://doi.org/10.1145/344949.344959

[39] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. 2008. How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 177–184. https://doi.org/10.1109/VLHCC.2008.4639081

[40] Brad A. Myers. 1991. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*. ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/120782.120805

[41] Mathieu Nancel and Andy Cockburn. 2014. Causality: A Conceptual Model of Interaction History. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 1777–1786. https://doi.org/10.1145/2556288.2556990

[42] Donald A Norman. 2010. *Living with complexity*. MIT press.

[43] Donald A Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic Books (AZ).

[44] Dan R. Olsen, Jr. 1984. Pushdown Automata for User Interface Management. *ACM Trans. Graph*. 3, 3 (July 1984), 177–203. https://doi.org/10.1145/3870.3871

[45] Dan R. Olsen, Jr. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[46] Stephen Oney, Brad Myers, and Joel Brandt. 2012. ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 229–238. https://doi.org/10.1145/2380116.2380146

[47] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: A Language and Environment for Expressing Interface Behavior. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/2642918.2647358

[48] Robert S Rist et al. 1986. Plans in programming: definition, demonstration, and development. In *Empirical studies of programmers*. 28–47.

[49] Gustavo Rovelo, Donald Degraen, Davy Vanacken, Kris Luyten, and Karin Coninx. 2015. Gestu-Wan - An Intelligible Mid-Air Gesture Guidance System for Walk-up-and-Use Displays. In *Human-Computer Interaction - INTERACT 2015*, Julio Abascal, Simone Barbosa, Mirko Fetter, Tom Gross, Philippe Palanque, and Marco Winckler (Eds.). Vol. 9297. Springer International Publishing, 368–386. https://doi.org/10.1007/978-3-319-22668-2_28

[50] Miro Samek. 2003. Who moved my state. *Dr. Dobb's Journal* (2003).

[51] Julia Schwarz, Jennifer Mankoff, and Scott Hudson. 2011. Monte Carlo Methods for Managing Interactive State, Action and Feedback Under Uncertainty. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 235–244. https://doi.org/10.1145/2047196.2047227

[52] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[53] Michael Terry and Elizabeth D. Mynatt. 2002. Side Views: Persistent, On-demand Previews for Open-ended Tasks. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST '02)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/571985.571996

[54] Davy Vanacken, Alexandre Demeure, Kris Luyten, and Karin Coninx. 2008. Ghosts in the interface: meta-user interface visualizations as guides for multi-touch interaction. In *Proceedings of the 3rd IEEE international workshop on Horizontal Interactive Human Computer Systems (TABLETOP '08)*. IEEE Computer Society, 81–84. https://doi.org/10.1109/TABLETOP.2008.4660187

[55] Jo Vermeulen, Kris Luyten, Elise van den Hoven, and Karin Coninx. 2013. Crossing the Bridge over Norman's Gulf of Execution: Revealing Feedforward's True Identity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1931–1940. https://doi.org/10.1145/2470654.2466255

[56] W. A. Woods. 1970. Transition network grammars for natural language analysis. *Commun. ACM* 13, 10 (Oct. 1970), 591–606. https://doi.org/10.1145/355598.362773