Standardizing the Representation of User Tasks

Martin R. Frank

University of Southern California - Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292-6695
frank@isi.edu

Currently, most demonstrational systems are built from scratch, typically by individuals, and are all quite unique and distinct from each other. It is still desirable to build more such small-scale systems in the future. However, the key to a larger-scale, collaborative and interdisciplinary effort is in defining a common representation for user tasks, and in defining a common input format for automation tools. This way, the Machine Learning community can focus on the inferencing while the Programming By Demonstration community can focus on the human interface for automating tasks, using the same representation.

This paper consists of two parts. The first section makes a case for spending some time at the symposium designing a common representation for user tasks and their automation. The second section briefly presents Grizzly Bear, a demonstrational tool that I have built over the last three years (and which taught me the importance of a well-defined underlying representation the hard way).

1 Position Statement or "What we should be spending our time on at the Symposium"

I believe that the key to a wider collaboration on automating tasks for users is in agreeing on a common format for internally representing these tasks. Therefore, I think it is worth spending some time at the symposium on this issue, and to maybe form a "standards sub-committee" that could keep working afterwards.

1.1 A Common Representation for User Tasks

The foundation for tools that constantly watch the user is a representation of user tasks or events.[1] The user task representation should be similar in spirit to Apple Events, and should provide for a hierarchical decomposition of events [2]. There are also task models for purposes other than user task automation [10]

This representation should be sufficient for tools that continuously watch the user because they have no user-visible interface until they detect repetition (e.g. Eager [1]).

However, there is also a place for automation tools that the user explicitly invokes and demonstrates to, especially for automating tasks that can then be invoked via a shortcut later (rather than having to re-perform the task and wait for the automation component to again pick up on it). For these tools, we should also agree on a common format for their input (their abstract user interface).

1. I will use these two terms synonymously. Task automation is concerned with low-level tasks (high-level events) such as "open file".

1.2 A Common Input Format for Explicitly Invoked Trace-Based Task Automation Tools

A common input format for explicitly invoked trace-based tools could be one "stimulus trace" and one "response trace", in the sense that the user is demonstrating that the stimulus will trigger the response. The user could provide multiple examples of this type for more complex tasks. There could also be the option of providing counter-examples which tell the automation tool when the response should *not* be triggered (counter-examples would require a stimulus trace only).

1.3 A Common Input Format for Explicitly Invoked Snapshot-Based Task Automation Tools

Snapshot-based demonstrational tools do not watch the user constantly (a video camera metaphor) but rather only look at particular snapshots of state that are pointed out by the user (a flashlight-photography metaphor). This approach avoids problems with users re-ordering events and inserting extraneous events. Examples of snapshot-based tools are GITS [3] and DEMO [4].

A possible common input format could consist of a *before* snapshot (context), a triggering event (stimulus), and an *after* snapshot (response). The event could be in the format discussed in Section 1.1, the state could consist of a list of objects holding domain-specific state relevant to end-user automation. As before, the user should probably be able to provide multiple such examples for more complex behavior, and there could again be the option of giving counter-examples as well.

2 Grizzly Bear - Lessons Learned

The rest of the position paper describes Grizzly Bear, a new demonstrational tool for specifying user interface behavior that I have built over the last three years as part of my dissertation. The section re-uses much of a two-page paper accompanying a "refereed demonstration" at UIST'95 [5].

For the purposes of this position paper, I should mention that Grizzly Bear uses a snapshot-based approach, and that its input format is similar to the one proposed in Section 1.3 ("negative examples" are identical to counter-examples). It is also worth mentioning that Grizzly Bear taught me the importance of a well-defined underlying representation the hard way - I spent nearly a year trying to build a demonstrational system without a precise definition of its input and output, got nowhere, and finally re-implemented from scratch after defining the underlying representations first!

Grizzly Bear is a tool for demonstrating user interface behavior. It can handle multiple application windows, dynamic object instantiation and deletion, changes to any object attribute, and operations on sets of objects. It enables designers to experiment with rubber-banding, deletion by dragging to a trashcan and many other interactive techniques.

There are inherent limitations to the range of user interfaces that can ever be built by demonstration alone. Grizzly Bear is therefore designed to work hand-in-hand with a user interface specification language called the Elements, Events & Transitions model. As designers demonstrate behavior, they can watch Grizzly Bear incrementally build the corresponding textual specification, letting them learn the language on the fly. They can then apply their knowledge by modifying Grizzly Bear's textual inferences, which reduces the need for repetitive demonstrations and provides an escape mechanism for behavior that cannot be demonstrated.

Grizzly Bear is the bigger brother of a previous system called Inference Bear [6,7]. The behavior that can be demonstrated to Grizzly Bear is a superset over that of the earlier system. The enhancements include conditional behavior as well as behavior exhibited by sets of objects.

This higher expressive power is achieved by incorporating *positive* and *negative* examples in addition to the *before* and *after* examples already used by Inference Bear. Figure 1 shows Grizzly Bear's user interface.

Giving one example to Grizzly Bear consists of working through its iconic buttons from left to right. One demonstration consists of one or more such examples. After each example, the corresponding textual inference is displayed, and the behavior can be tested interactively by temporarily going into a test drive mode.

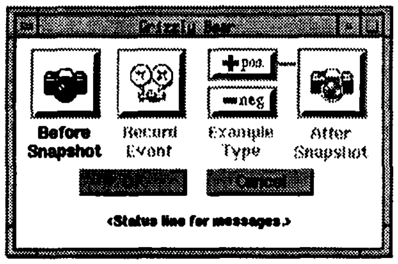

Figure 1. Grizzly Bear's Control Panel.

The use of Grizzly Bear is best explained by examples. The rest of this paper shows how to build a small "file manager" application which lets users create and delete documents and folders. Figure 2 shows its user interface, which is laid out using a conventional user interface builder [8].

The behavior to be demonstrated is that new folders and documents can be created on the main canvas by dragging from their prototypes. They can then be moved around, and they can be deleted by dragging them to the trashcan.

I will discuss creation by dragging in detail. Assume the designer decides that pressing the mouse down on the *Folder* icon makes a new red folder appear under the mouse, that the new folder then tracks the mouse, and that releasing the mouse button stops the tracking and makes the folder black.

Showing this functionality consists of one demonstration each for the *press*, *motion* and *release* events on the *Folder* icon. (All events between a *press* and a *release* event are received by the element which received the *press* event.)

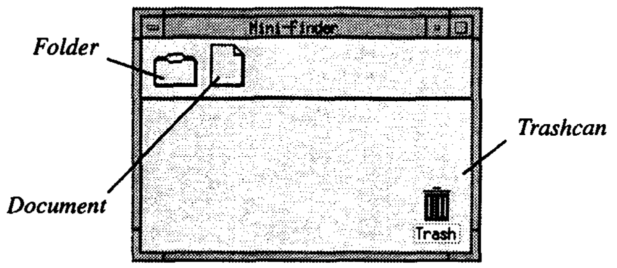

Figure 2. Layout of the Mini-Finder Application.

The first demonstration shows Grizzly Bear that pressing down on the *Folder* icon makes a new red folder appear there (Figure 3). The *before* snapshot is on the left, with a feedback icon representing the triggering event superimposed. (The feedback icons are borrowed from Marquise [9].) The *after* snapshot is on the right. In this case, the *after* snapshot shows that a red copy of the prototype folder is created. (The red color appears grayish in the screen shot, and is hard to see - the new folder is located on top of the old one.)

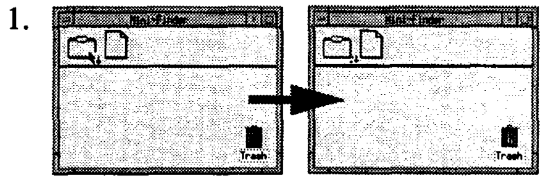

Figure 3. Drag-and-drop: the *Press* Event.

If we were to test the behavior now, we could create a new folder. However, it would not follow the mouse but rather appear over the prototype folder, obscuring it.

So let us show that the newly created folder icon then follows the mouse. This can be done by demonstrating that red objects follow the mouse, as the red color is one way of telling the newly created folder apart from the others. We demonstrate this behavior by introducing two temporary folder icons in the first *before* snapshot, and by then demonstrating that the red one follows the mouse (Figure 4). This demonstration consists of two examples.

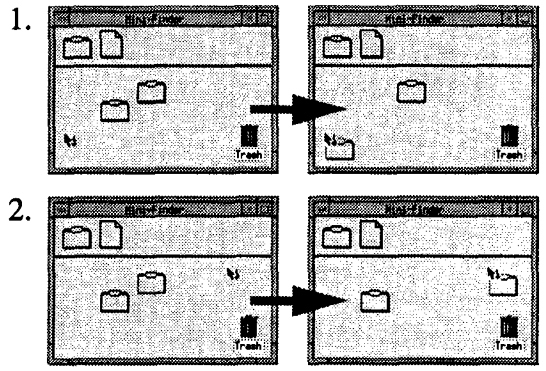

Figure 4. Drag-and-drop: the *Motion* Event.

If we test the current behavior now, we can create the first folder as desired, but it appears in red on the canvas. If we create further new folders, we will drag the previous folders as well as the new ones! This is because Grizzly has inferred that all red objects track the mouse, and because all folders remain red after their creation so far.

Let us thus complete the example by demonstrating that the color of newly created folders reverts to black upon the *release* event of the drag-and-drop interaction (Figure 5).

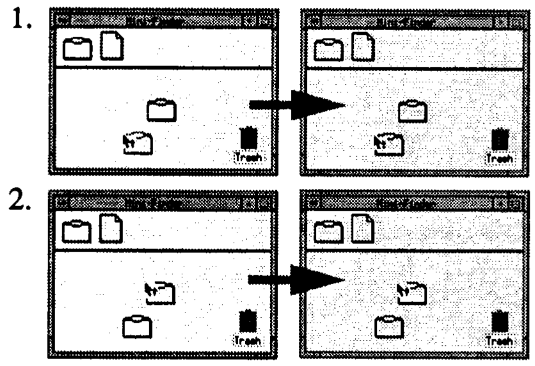

Figure 5. Drag-and-drop: the *Release* Event.

If we test the current behavior now we can indeed create folders by dragging from the palette as intended.

I will only outline how the remaining functionality is demonstrated. First, we want to be able to create documents in the same way as folders. We can achieve this by repeating the demonstrations above for documents, or we can make copies of the generated textual specifications for folders, and adapt them for documents. (For the sake of completeness, we could also generalize the synthesized specifications to work for both folders and documents, or we could have demonstrated the more general behavior in the first place. However, these solutions require more sophistication on the designer's part.) The remaining functionality concerns moving and deleting objects. These demonstrations make use of negative examples for the first time in this paper. We show that canvas objects follow the mouse (positive examples) while the prototype objects and the trashcan do not (negative examples). Similarly, in the final demonstration, releasing canvas objects over the trashcan deletes them (positive examples), but this is not true for the prototype objects (negative examples).

Acknowledgments

Brad Myers contributed the feedback icons for triggering events. The camera and recorder icons were designed by Kevin Mullet while at Sun Microsystems.

References

[1] A. Cypher. "EAGER: Programming repetitive tasks by example." In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pp. 33–39, (New Orleans, LA, Apr. 28-May 2) 1991.

[2] D. Kosbie and B. Myers. "Extending programming by demonstration with hierarchical event histories." In *Fourth International East-West Conference on Human-Computer Interaction*, pp. 147–157, (St. Petersburg, Russia, Aug. 2-5) 1994.

[3] D. Olsen and K. Allan. "Creating interactive techniques by symbolically solving geometric constraints." In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 102–107, (Snowbird, UT, Oct. 3-5) 1990.

[4] D. Wolber and G. Fisher. "A demonstrational technique for developing interfaces with dynamically created objects." In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 221–230, (Hilton Head, SC, Nov. 11-13) 1991.

[5] M. Frank. "Grizzly Bear: A demonstrational learning tool for a user interface specification language." In *Proceedings of the ACM Symposium on User Interface Software and Technology*, (Pittsburgh, PA, Nov. 15-17) 1995.

[6] M. Frank and J. Foley. "A pure reasoning engine for programming by demonstration." In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 95–101, (Marina del Rey, CA, Nov. 2-4) 1994.

[7] M. Frank, P. Sukaviriya, and J. Foley. "Inference Bear: Designing interactive interfaces through before and after snapshots." In *Proceedings of the ACM Symposium on Designing Interactive Systems*, pp. 167–175, (Ann Arbor, MI, Aug. 23-25) 1995.

[8] T. Kuehme and M. Schneider-Hufschmidt. "SX/Tools - An open design environment for adaptable multimedia user interfaces." *Computer Graphics Forum*, vol. 11, pp. 93–105, Sep. 1992.

[9] B. Myers, R. McDaniel, and D. Kosbie. "Marquise: Creating complete user interfaces by demonstration." In *Proceedings of INTERCHI, ACM Conference on Human Factors in Computing Systems*, pp. 293–300, (Amsterdam, The Netherlands, Apr. 24-29) 1993.

[10] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Sacher. Declarative interface models for user interface construction tools. In *IFIP Working Conference on Engineering for Human-Computer Interaction*, (Grand Targhee Resort, WY, Aug. 14-18) 1995.