

Inference Bear: Designing Interactive Interfaces through Before and After Snapshots

Martin R. Frank Piyawadee “Noi” Sukaviriya James D. Foley
{martin, noi, foley}@cc.gatech.edu

Graphics, Visualization & Usability Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

ABSTRACT

We present Inference Bear (“An Inference Creature based on Before and After Snapshots”) which lets users build functional graphical user interfaces by demonstration.

Inference Bear is unique in its use of a domain-independent reasoning engine. This approach has several advantages over systems that are closely tied to their domains. Most notably, Inference Bear reasons about a class of relationships that is defined by their computational complexity while rule-based systems are limited to reasoning about the class of relationships that the designer foresaw when building the system.

However, it is also more difficult to design domain-independent demonstrational systems that are as easy to use as their domain-specific counterparts. The paper addresses this issue, and other issues relating to domain-independence.

KEYWORDS: Rapid prototyping, human-computer dialog specification, programming by demonstration.

INTRODUCTION

This paper will focus on the interactive design process when using Inference Bear, as is appropriate for this conference. Inference Bear is built on top of a reasoning engine whose computational aspects we have described in a previous paper [6]. Using an existing reasoning engine puts constraints on the user interface of the demonstrational tool while also leaving many degrees of freedom. We will quickly describe the input and output of the engine we are using, and then explain how we built Inference Bear on top of it throughout the paper.

The input to the engine consists of a description of a change of state, and of a description of when this change of state should occur at run-time. The description of how the state changes consists of snapshots of “before” and “after” states.

Cite as: 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*, pages 167-175, (Ann Arbor, Michigan, August 23-25) 1995.

The description of when the state changes consists of a parameterized event. The input can consist of a single such example or of multiple examples.

The output of the engine consists of an algorithmic description of how the state changes in response to the event. How the state changes can depend on the previous state and on the event parameters.

Using this engine implies that the demonstrational system must provide a way for the user to input snapshots and events, but there are many possible ways of doing so. Important issues

in designing a user interface to the engine are how and when the users receive feedback on their demonstrations, how they specify events and their parameters, when they specify the event, and how they specify demonstrations consisting of multiple examples.

PROGRAMMING BY DEMONSTRATION

We shortly review related work on demonstrational systems. Peridot [10] supports designing scrollbars, buttons, choice boxes and similar objects by demonstration. Lapidary [12] focuses on creating application-specific objects. Metamouse [9] learns graphical procedures by example. Druid [13] lets designers attach simple functionality to graphical user interfaces, such as enabling, disabling, hiding and showing buttons. Eager [1] watches users perform operations and detects and automates repetition. Mondrian [in 2] is a graphical editor that can learn new composite commands by demonstration. DEMO [15,3] uses a stimulus-response paradigm for demonstrating the behavior of graphical objects. Chimera [8] infers constraints on the movement of objects given multiple snapshots. Finally, Marquise [11] uses domain knowledge in order to support building graphical editors.

All of these systems use by-demonstration techniques but they are not easily compared because they have different goals and use different techniques. We make an attempt to classify them in Table 1 nevertheless.

The first two columns describe user interface aspects. The first column shows if the system is constantly watching the user during normal operation or if it is explicitly invoked. The second column lists if the system asks the user for con-

	<i>Interface</i> Is Eager (Constantly Watches User)	<i>Interface</i> Feedback about Inference Process	<i>Capabilities</i> Run-time Object Instantiation	<i>Capabilities</i> (Subjective) Strength in Geometric Relations	<i>Internals</i> Search Space Reduction	<i>Internals</i> Is Rule-Based	<i>Internals</i> Temporary Behavior Storage	<i>Internals</i> Inferencing Result
Peridot [10] 1987	Yes	Clarification	Partially ^a	Low	Selecting (Optional) ^b	Yes	Snapshots	One-Way Constraints
Lapidary [12] 1989	No	Clarification	No	Low	None ^c	No	Snapshots	One-Way Constraints
Metamouse [9] 1989	Yes	Clarification, Prediction	Yes	Medium	Touch Sensitive ^d	No	Event Recording	Graphical Procedure
Druid ^e [13] 1990	No	Clarificaton	No	not applicable	None ^c	No	Event Recording	Script
Mondrian [in 2] 1991	Yes (in teach- ing mode)	Synthesized Speech	Yes	Medium	None ^c	Yes	Event Recording	Macro ^f
Eager [1] 1991	Yes	Prediction	not applicable	not applicable	not applicable	No	Event Recording	Macro
DEMO [15,3] 1991/92	No	Clarification	Yes	High	Auxiliary Objects	Yes (DEMO II)	Compressed Snapshots	Response Description
Chimera ^g [8] 1991	No	None ^c	No	High	Auxiliary Objects ^h	No	Snapshots	Two-Way Constraints
Marquise [11] 1993	No	(optional) Clarification	Yes	Low	None ^c	Yes	Event Recording	LISP Code
Inference Bear 1995	No	Script Display	Yes	Medium	Motion Sensitive	No	Ev. Rec. and Snapshots	Script ("Transition")

Table 1. Overview of Demonstrational Systems.

- a. Peridot can dynamically create objects such as popup menus but cannot copy (instantiate) objects at run-time.
- b. The system gives the designer the option to explicitly reduce the search space by selecting relevant objects.
- c. Note that "none" should be interpreted as "none published".
- d. Metamouse allows to describe relationships between distant objects through auxiliary objects.
- e. Note that only Druid's demonstrational component is discussed here but not its rule-based design assistant.
- f. Macro invocation is depicted as a miniaturized Before and After snapshot, and the body is depicted as a graphical storyboard.
- g. Note that only Chimera's "inferring constraints from snapshots" subsystem is discussed here.
- h. Chimera also uses a variety of other techniques to reduce the search space.

firmation and clarification after each inference. Some systems query the user when they make an inference (marked with "Clarification" in the table), others indicate their inferences by displaying their prediction of what the user is doing next ("Prediction"). Most inferencing systems will sometimes guess wrong - the clarification dialog gives the user an opportunity to correct and fine-tune inferred behavior. The disadvantage is that going through this clarification process after every inference can be distracting.

The next two columns make an attempt to measure the capability of the system. The third column indicates if the prototypes can handle the creation and deletion of graphical objects at run-time. The fourth column indicates to what degree the systems can infer geometric relationships between objects. This classification is inherently subjective because the design goals of these systems are different (for example, Eager does not attempt to infer geometric relationships) and because their capabilities are different (for example, Eager is "stronger" than Inference Bear in being able to deal with

repetition, Inference Bear is "stronger" in inferring geometric relationships). We labelled systems which can detect simple relationships such as centering and aligning "Low", systems which can detect more general relationships "Medium" and the most sophisticated systems "High". This, again, is a crude and necessarily subjective classification.

The remaining columns are concerned with the implementation of the inferencing systems. The fifth column describes how the system reduces the number of objects that it checks for relationships. Some systems use auxiliary objects such as guide wires to let the user specify the relevant objects and their relationship. Inference Bear limits the number of objects it considers to those that change during the demonstration [6]. The sixth column describes if the inferencing is based on rules or on an algorithm. The advantage of rule-based systems is that they can encode knowledge about common use of the system. The disadvantage is that the rules can sometimes miss even simple relationships while algorithm-based systems can handle all relationships within

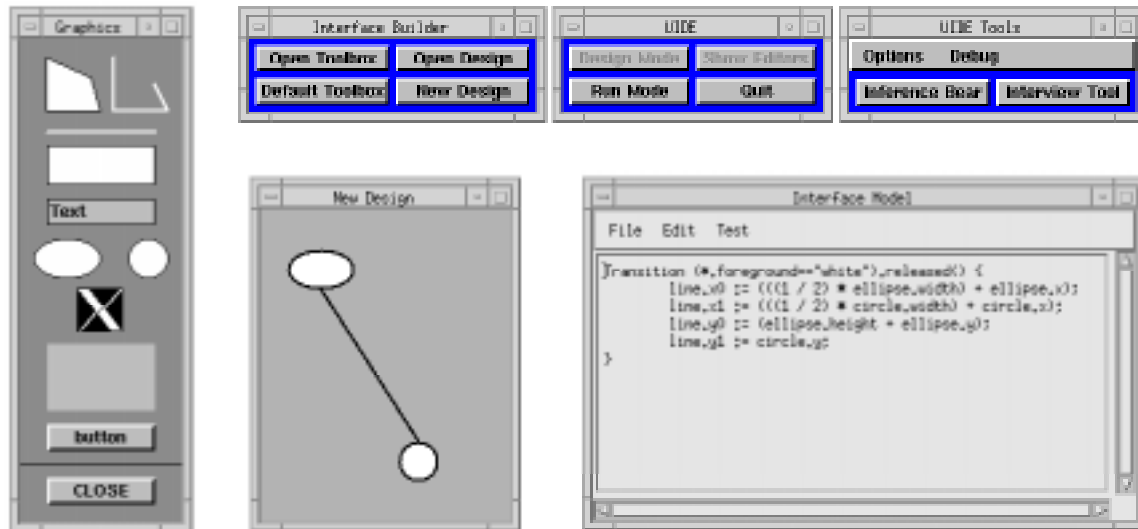


Figure 1. Overview of the Design Environment.

a certain class. The seventh column describes how previous demonstrations are captured during the inference process. There are two main approaches towards capturing demonstrations. Event-recording stores the events during a demonstration by the user while snapshot-taking records a series of states. The last column describes the result of the inferencing process.

THE DESIGN ENVIRONMENT

Figure 1 shows an overview of our design environment. The three small windows are the control panels of its main components.

The left-hand control panel belongs to the interface builder we use [7]. It lets you open user interface designs and palettes of reusable interface elements (“toolboxes”). One such toolbox is shown on the left, labelled “Graphics”. The current user interface design is shown in the center, consisting of an ellipse and a circle which are connected by a line.

The middle control panel belongs to UIDE, a descendant of the original User Interface Design Environment [4]. In the context of this paper, it suffices to know that it lets you open the text editor shown on the right-hand side of Figure 1, and that it can switch the interface to run mode by reading and executing the specification in this editor. The “Interface Model” specifies the behavior of the interface at run time. In Figure 1, it specifies how the line changes when a mouse button is released over an object with a white foreground color.

The “UIDE Tools” control panel provides the interface to advanced tools in the UIDE environment. The “Interview Tool” helps the designer fill in the textual model by asking questions about the current user interface layout. It is described in [5] and will not further be discussed here. The “Inference Bear” is the tool we are concerned with - it lets designers describe the behavior of the user interface by demonstration. For example, it can infer the body of the transition shown in Figure 1.

INFERENCE BEAR

Figure 2 shows the control panel that appears when the user clicks on the “Inference Bear” button.

Giving one example consists of working through the four iconic buttons from left to right. The designer first sets up for the *before* snapshot by editing the current design using the interface builder. She then clicks on the left-most button to tell the system that she is done with editing.

The next step is to tell the system what kind of event triggers the behavior. Inference Bear deals with very fine-grained events from the windowing system, such as *press*, *motion*, *release*, *enter* and *leave* events. This allows the designer to use Inference Bear to describe highly interactive techniques such as rubberbanding and cursor-dependent object highlighting.

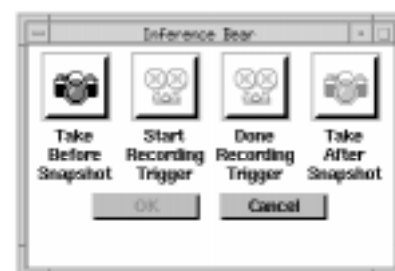


Figure 2. Inference Bear.

However, it also poses an interesting problem in recording the triggering event because there may be no way to cause this event without also causing extraneous events. For example, if the desired trigger is pressing a mouse button over an object then there may be no way to get to this object without causing at least an *enter* event on the object’s layout parent and several mouse motion events; and there will be more events after pressing the mouse button, such as the corresponding *release* event and more mouse motion

events. Which event is the one that triggers the behavior to be demonstrated?

There are several solutions to this problem. Peridot [10] uses a “simulated mouse” which is a window depicting a mouse including the state of its buttons. The system can then use the state and position of the simulated mouse, freeing the real mouse for meta-level commands. DEMO [15] always records a *press* event and then asks the user whether she really meant *press* or one of three other event types (*release*, *enter* and *leave* in our terminology).

However, both are “indirect” solutions which are not as natural as causing the same event that will trigger the run-time behavior. We use an alternative technique which uses time to distinguish the triggering event from other events. Figure 3 shows Inference Bear’s control panel just after the “Take Before Snapshot” button was clicked.

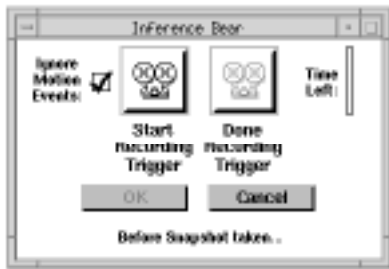


Figure 3. Capturing Events with Inference Bear.

The “Take Before Snapshot” and “Take After Snapshot” buttons of Inference Bear’s control panel are temporarily replaced with two controls for recording the event. When the user clicks on the “Start Recording Trigger” button, Inference Bear records user events for five seconds and then uses the last event that was recorded before time ran out. The “Time Left” bar proportionately shrinks during the five seconds and Inference Bear beeps similar to a camera in “automatic timer” mode (five short beeps followed by one long beep). The “Ignore Motion Events” checkbox is left checked unless one is actually recording a motion event.¹

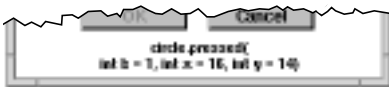


Figure 4. Feedback on Recorded Events.

Figure 4 shows part of the control panel after the event was recorded (*b* indicates which mouse button was pressed, *x* and *y* indicate the location of the click). The designer can then either re-record the event (if she recorded an erroneous event) or go on by clicking the “Done Recording Trigger” button. Inference Bear will prevent the designer from going on when a mistake was made (such as not recording any

1. Otherwise, you would need a steady hand when recording e.g. a *press* event or you will cause trailing mouse motion events.

event or recording a different type of event than in previous examples of the same demonstration), and display an explanation.

The final step is to tell the system what should happen if the interface is in the *before* state and the recorded event occurs. This is done by bringing the user interface design to the state it should go to, and by pressing the “Take After Snapshot” button to let the system know that one is done.

The system now responds by running the inference engine on this example and by showing the resulting transition in the editor. The interface design is reset to the state it was in before it was edited for the *before* snapshot.² The designer then tests if the inferred behavior is what she had in mind (by either reading the text of the transition or by going to Run mode and testing it interactively). If it is acceptable, she clicks “OK” in Inference Bear’s control panel which causes the inferred behavior to become part of the overall interface model. If it is not what she had in mind, she proceeds by giving more examples in the same manner. The inference engine is then called using the old examples plus the new ones.

EXAMPLE 1: INTRODUCTORY EXAMPLE

Let us explain this dialog between the designer and Inference Bear using a simple example. The designer has created a user interface consisting of a window containing a single button. The behavior to be inferred is that the button moves one button length to the right every time it is clicked.

She provides the first example shown on the left-hand side of Figure 5. We have superimposed the *before* and *after* snapshots so that the screenshot shows the *after* snapshot while the dotted shadow indicates where the button was located in the *before* snapshot. The recorded event is a click on the button. The system responds with the script shown in the editor.

Inference Bear’s initial conjecture here is that the button moves to the absolute position shown in the *after* snapshot. This behavior shows that our use of a domain-independent reasoning engine is *not* transparent to the user. A domain-dependent demonstrational system could have a rule such as “if objects move and they are placed next to where they were then infer a script that implements a relative move”. If such a rule exists the system can infer the solution from a single example.

Our reasoning is based only on the types and values of attributes but not on their names [6]. A demonstration consisting of a single example will always be solved by assigning constants to the variables. The engine uses the simplest solution to a demonstration because there generally is an infinite number of more complex solutions to a demonstration,

2. It is interesting to note that we simply use a different instantiation of the inference engine for resetting. We take a snapshot before the design is first edited for the *before* state and a snapshot after the editing for the *after* state, and then let this engine instantiate compute and execute a transition which transforms the interface back to the former state.

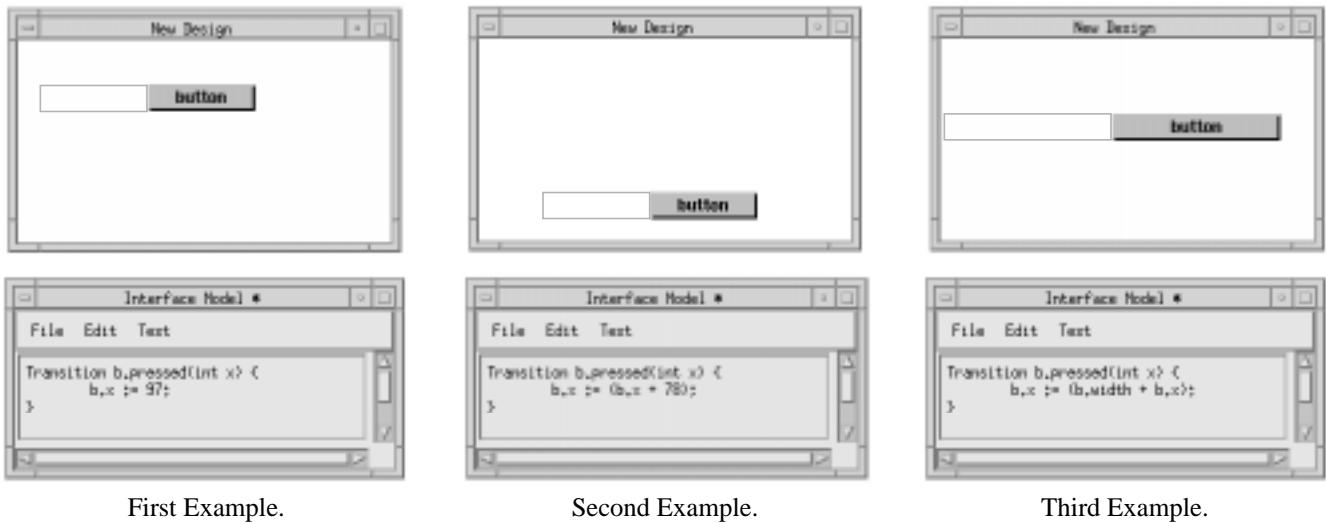


Figure 5. Inference Bear's successive inference refinement.

and there is no way of choosing one over the other without domain knowledge.

The designer now gives a second example as shown in the middle column of Figure 5. She has moved the button to a different position for the before snapshot and shown that the button again moves to the right. Inference Bear responds by refining the inference as shown. It has now inferred that the new button position is relative to the old position.

However, it has inferred that the button moves by a fixed amount of pixels rather than by one button length. This behavior shows another user-visible impact of using the domain-independent engine - the engine's vision is based on motion. This technique allows the engine to prune its search space without resorting to domain knowledge. It eliminates all attributes that did not move (change) during a demonstration, thus enabling it to spend more computational resources on the remaining attributes. So far, the width of the button has never changed so that the engine assumes that it is of no relevance to the demonstration.

The desired solution is found after the designer gives the third example shown in the right column of Figure 5.

How many examples are needed depends on how "good" the examples are. Poor examples are examples that are too similar to previous examples. For example, had the designer demonstrated the first example of Figure 5 twice then Inference Bear would still have solved the demonstration using a constant. Good examples are examples that invalidate the previous solution while being consistent with the desired solution. In Figure 5, the second example invalidates the Bear's hypothesis that the button moves to an absolute position, and the third example invalidates its hypothesis that the button moves by a constant offset.

EXAMPLE 2: OBJECT INSTANTIATION AND DELETION

This section will introduce the underlying textual language in some detail as this clarifies what kind of interfaces can be

built using our environment. We call this specification language the Elements, Events & Transitions model.

Three simple transitions are shown in Figure 5. Transitions can create and delete elements in addition to changing their attributes, and Inference Bear can parameterize the creation of objects by computing attributes of newly created objects in the same way as it computes attributes of other objects. Transitions create new elements by copying from prototype elements. A statement of the form "object *o* := create *rectangle* on *canvas*" reads "create an object by copying *rectangle* onto *canvas* and let the rest of the transition refer to the new object as *o*". An object can serve as a prototype for a new object if it is of the same type and if it differs in no more than ten attributes. Inference Bear will use the most similar existing object as the prototype for a new object (the one with the fewest differing attributes).

Consider the interface shown in Figure 6. The task is to demonstrate the following behavior. Clicking on an object in the palette causes the clicked-upon object to become highlighted and the other object to be de-highlighted. Clicking on the canvas causes a circle to appear if the circle in the palette is highlighted. Pressing the mouse on the canvas causes a new line to appear if the line in the palette is highlighted. One end of the line then follows the mouse until the mouse button is released. Dragging an object over the trashcan causes the trashcan to highlight in order to indicate that the object can be dropped there. Dropping objects on the trashcan deletes them.

This behavior can be shown to Inference Bear in ten demonstrations. It is, however, currently necessary to tune some of the inferred transitions by hand (we will discuss under which circumstances this is appropriate later). We show the complete interface model for this behavior below. It consists of ten transitions.

A transition consists of a header which specifies when the transition gets executed, and a body which specifies how its execution changes the state. The simplest headers consist of

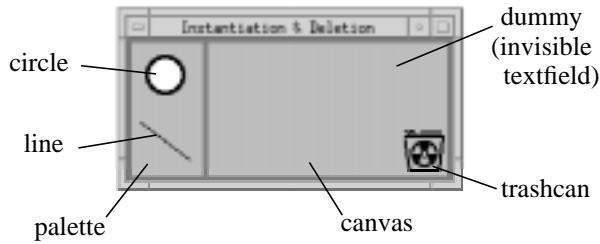


Figure 6. Instantiating Palette Objects on a Canvas.

the name of a concrete object and an event, such as “triangle.pressed()” which gets executed if the user presses any mouse button over the object *triangle*. More complex headers can restrict the execution to specific situations. For example, “(*.name==triangle && flag.color==”red”) .pressed()” is applicable if a mouse button is clicked over *triangle*, but only if the object *flag* is currently red. It is also possible to restrict the applicability based on the parameter values. For example, “triangle.pressed(int b<=2)” is applicable only if the left or middle mouse button is pressed, assuming that a three-button mouse is used, and that an ordinal parameter *b* is used to communicate which mouse button was pressed.

```
Transition circle.pressed()
{
    circle.lineWidth := 3;
    line.lineWidth := 1;
}
Transition line.pressed()
{
    line.lineWidth := 3;
    circle.lineWidth := 1;
}
```

These two transitions describe that clicking on the palette objects toggles their selection status. It is up to the designer to decide how the selection status is displayed to the user. We have used the line width of objects here.

```
Transition ((*.name=="canvas") &&
           (circle.lineWidth==3)).pressed(int x, int y)
{
    object n := create circle on canvas;
    n.lineWidth := 1;
    n.x := x;
    n.y := y;
}
```

This transition above describes how a new circle is created by clicking on the canvas. Inference Bear can infer the body of this transition but the designer currently has to add that this behavior should only occur if the circle in the palette is highlighted (&& *circle.lineWidth==3*).

```
Transition ((*.name=="canvas") &&
           (line.lineWidth==3)).pressed(int x, int y)
{
    object n := create line on canvas;
    n.lineWidth := 1;
    dummy.text := n.name;
    n.x0 := x;
```

```
n.y0 := y;
n.x1 := x;
n.y1 := y;
```

```
}
```

Similar to the previous transition, the one above describes that pressing a mouse button over the canvas will create a line there if the line in the palette is highlighted (the new line will first look like a dot). The *dummy* object is a - normally invisible - text field that holds the name of the currently created line. It is used as a global variable whose value can be demonstrated by editing the text field.

```
Transition ((*.name=="canvas") &&
           (dummy.text!="")).motion(int x, int y)
{
    (*.name==dummy.text).x1 := x;
    (*.name==dummy.text).y1 := y;
}
```

This transition causes one end of the line to follow the mouse when a line is currently being created. Making the behavior dependent on the *dummy* field again has to be added by hand (&& *dummy.text!= ""*). Inference Bear can infer the two set expressions in the body (set reasoning will be explained in Example 4).

```
Transition ((*.name=="canvas") &&
           (line.lineWidth==3)).released(int x, int y)
{
    dummy.text := "";
}
```

The transition above resets the dummy text field once the user releases the mouse after creating a new line.

```
Transition ((*.class=="Line") &&
           (*.name!="line")) ||
           ((*.class=="Circle") &&
           (*.name!="circle")).motion(int x, int y)
{
    self.x := x;
    self.y := y;
}
Transition ((*.class=="Line") &&
           (*.name!="line")) ||
           ((*.class=="Circle") &&
           (*.name!="circle")).released(
           string o == "trashcan")
{
    delete self;
}
```

The first transition implements dragging for all circles and lines besides the palette objects themselves. The second transition describes that if such objects are released over the trashcan they are deleted. The comparison *o== "trashcan"* is a parameter constraint - the body of the transition is only executed if the constraint holds. The *self* keyword refers to the object that matched the set expression in the header. The designer currently has to generalize from concrete objects to *self* by hand, similar to generalizing from concrete events to more complex transition invocations.

```
Transition trashcan.entered(int b != 0)
{
```

```

trashcan.fillForeground := "red";
}
Transition trashcan.left(int b != 0)
{
trashcan.fillForeground := "black";
}

```

Finally, the trashcan highlights in red when it receives enter and leave events. These two transitions are executed only if a mouse button is being held (parameter constraint $b \neq 0$).

There are many other ways of creating new objects with Inference Bear. Here is an outline of how dragging from a palette can be implemented. The *press* event on the circle in the palette makes an instant copy of the circle. The copy then stays behind on the palette as “circle”, while the new circle receives an automatic name and follows the *motion* events. A *release* event on the canvas then places the circle there, while such events on other canvasses are ignored.

EXAMPLE 3: USER-DEFINED ATTRIBUTES

Some attributes such as x , y , *width* and *height* are common to all graphical objects, while others are common to all objects of a certain type, such as *text* for text fields. The designer can also attach custom attributes. This is convenient in a variety of situations. For example, one can implement two different types of selection this way. That is, one can give objects the boolean attributes “primarySelection” and “secondarySelection”.

The advantage of InferenceBear here is that it can immediately infer transitions that use these new attributes. For example, clicking the left mouse button could toggle the primary selection which is indicated by turning the object red, and clicking the middle mouse button could toggle the secondary selection which is indicated by turning the object’s border fat. Other behavior can then be dependent on the selection status of objects. Reasoning about user-defined attributes would not be possible if Inference Bear relied on attribute names to draw inferences [6].

EXAMPLE 4: SET REASONING

Inference Bear can infer set expressions such as “objects that are higher than two hundred pixels” or “objects that are not red” on the left-hand side of assignments. Inference Bear looks for set expressions only if it cannot solve the demonstration using conventional assignments.

In this example, we want to show Inference Bear that clicking on the Left-Aligner will align the left sides of the currently selected objects to the left side of this button. The dotted white shadows in Figure 7 again indicate where the moved objects were located in the *before* snapshot.

A summary of the set reasoning process is that after a conventional solution cannot be found the reasoning engine tries to find a relationship between the variables that changed between the *before* snapshots (called “source” variables) and the variables that changed from at least one *before* to a corresponding *after* snapshot (called “target”

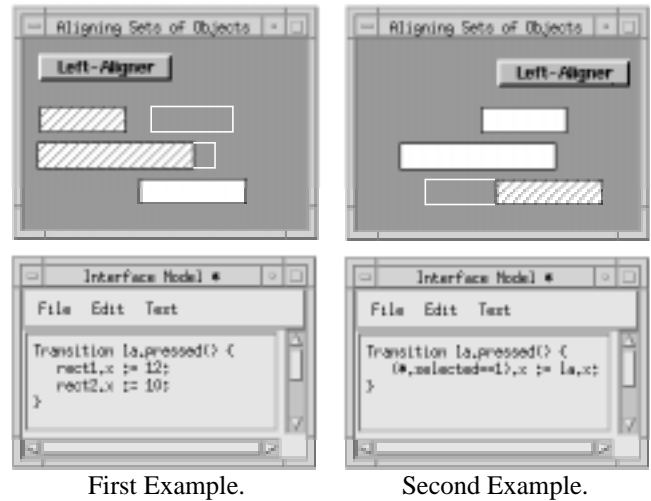


Figure 7. Aligning Sets of Objects.

variables). The (slightly idealized) input to the set reasoner for this case is as follows.

Boolean	Source Variable	rect1.selected	[1 0]
Boolean	Source Variable	rect2.selected	[1 0]
Boolean	Source Variable	rect3.selected	[0 1]
Integer	Source Variable	la.x	[10 90]
Integer	Target Variable	rect1.x	[10 -]
Integer	Target Variable	rect2.x	[10 -]
Integer	Target Variable	rect3.x	[- 90]

The n th value in brackets denotes the value of the variable in the n th *before* snapshot (for source variables) or n th *after* snapshot (for target variables). A hyphen indicates that a value remained unchanged in an example. The set reasoner then tries to find an expression that yields true when the values change. In this example, the comparison “selected==1” is true for each object when its x coordinate changes.

SHOULD THE DESIGNER BE EXPOSED TO A TEXTUAL LANGUAGE?

Inference Bear differs from other demonstrational systems in that it does *not* shield the designer from a textual language that describes interface behavior. Is this appropriate?

Clearly, it is desirable to provide a static, editable description of user interface behavior. This static description may also be a visual language. Unfortunately, we have not been able to identify a readily understandable visual language that is equivalent in power to the Elements, Events & Transitions model. The most difficult problem there is in describing the more complex conditional transitions.

Our subsequent goal was thus to combine a by-demonstration approach with textual editing. This is done by exposing the designer to the textual equivalent of the inferred transitions from the start. At this point, she does not have to understand the transitions - she can also test the behavior at any time by going to Run Mode, even during a demonstra-

tion - but she can glance over them and will eventually become more familiar with the textual form.

A typical use of Inference Bear by more experienced users is to create new transitions by one, maximally two, examples and by then modifying the generated output manually. One measure of the usefulness of Inference Bear even to the most experienced designers is that the first author, arguably the ultimate “power user” of the environment, himself prefers this hybrid approach to inputting the text directly - the hybrid approach is less error-prone and simply faster than typing the text from scratch.

We feel that a design environment where all behavior *has* to be described by demonstration will always remain a toy. An environment which combines the relative strengths of demonstrational and textual editing allows new users to quickly get started without frustrating the more experienced users.

BIASING THE INFERENCE TOWARDS SPECIFIC APPLICATIONS - BUG OR FEATURE?

One of Inference Bear’s design rationales is to keep its reasoning independent of its domain. But why is this desirable? Shouldn’t demonstrational systems be as domain-specific as they can possibly be?

For example, making an object follow the mouse will always take two examples with Inference Bear. We could take advantage of the domain knowledge we have and build in a rule that checks if the mouse position is close to the position of a moved object. This way, we could guess what the designer is demonstrating from a single example. Wouldn’t we do her a favor?

In our view, the answer to this question strongly depends on how large the class of situations is that the demonstrational system can handle.

A demonstrational system which specializes in a reasonably small-scale task such as aligning objects is well served by building in as much knowledge about aligning as possible. If we have detailed knowledge about what the designer is doing and if what she can do is fairly restricted (say, either left-align, center, or right-align objects) then the demonstrational system can quickly pick up on demonstrations, and single examples are sufficient.¹

We believe that demonstrational systems which address large-scale tasks such as building graphical applications by demonstration are better served by building in as *little* domain knowledge as possible. Constructing a complex graphical user interface nearly requires the power of general-purpose programming. That is, the domain is so large that the little knowledge we could build in will likely do more harm than good because it makes the inferring less predictable.

1. Note, however, that if the domain is *too* narrow we may be better off by not using a demonstrational technique at all, of course.

USABILITY TESTING

Observing users is the litmus test of any demonstrational system. We have informally tested Inference Bear in usability experiments. Subjects were given the task of demonstrating simple transitions similar to the ones in Example 1 and 4. They were not given instructions or assistance other than starting up Inference Bear for them and showing them an example use of Inference Bear. Most subjects were members of the Graphics, Visualization and Usability Center so that they represent a technical audience. There were a dozen subjects which used the system for about an hour and a large number of casual users which have used it for a few minutes (during the Center’s monthly demo days). The testing was not thorough enough to draw statistically significant conclusions but we will present some observations.

- The first observation is that demonstrational systems are potentially, but in no way inherently, easy to use. It took many, often seemingly small, refinements to Inference Bear until we achieved acceptable performance.
- It takes about twenty minutes to specify how two buttons change a window color (the interface consisted of two buttons named “red” and “green”, the users’ task was to demonstrate that clicking on those buttons sets the color of the window). About half of this time is spent figuring out how to use the interface editor rather than the demonstrational methodology.
- Users have little trouble with demonstrations consisting of a single example. Single-example demonstrations involving the creation or deletion of objects do not seem to be more difficult than others.
- Demonstrations involving two or more examples are significantly more difficult. It takes about fifteen more minutes to do the “moving button” task (Example 1), and all of this time was spent trying to understand how to demonstrate to Inference Bear. However, subjects could complete the remaining tasks quickly once they learned how to give multi-example demonstrations.

FUTURE WORK

Inference Bear is better in inferring the body of transitions than it is in inferring the invocation conditions of transitions. One solution is a companion for Inference Bear tentatively called the Expression Finder. This component will take examples consisting of a *before* snapshot, an event and a flag that indicates if this is a positive or a negative example. Its output is a transition header that will be invoked for situations such as the ones given in the positive examples. Much of this reasoning is already implemented in Inference Bear, such as finding the minimal difference between snapshots, and constructing set expressions. The Expression Finder’s tentative user interface is shown in Figure 8.

All the transitions in the preceding four examples are highly specific to their particular user interfaces. In the second-generation User Interface Design Environment [14], functionality at this level is captured in the *interface model*. UIIDE offers an optional higher level of abstraction which is called the *application model*. It consists of the same data de-

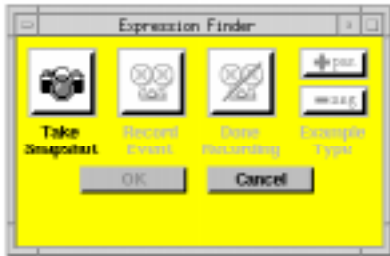


Figure 8. The Expression Finder, a future companion for Inference Bear.

scription and control description constructs as the lower-level interface model (“elements” and “transitions”). We want to eventually extend Inference Bear and the Expression Finder to be able to infer transitions at this level of abstraction as well. The main obstacle is that UIDE currently lacks an editable visualization of elements at the application model level.

CONCLUSION

We feel that Inference Bear shows that it is possible to build an easy-to-use Programming By Demonstration system using domain-independent reasoning, and that doing so has significant advantages over tightly coupling the reasoning to the domain, including the ability to reason about user-created entities and the ability to reason about entities at any level of abstraction.

We also feel that the user interface to Inference Bear (Figure 2) is sound and simple, and that it facilitates a playful, explorative design process that makes it easy to quickly come up with alternative user interface designs.

ACKNOWLEDGEMENTS

We would like to thank Schlumberger, Siemens, Sun Microsystems, and US West for sponsoring part of this research, and Christie Gerlach and Brad Myers for their feedback. The camera, recorder and radioactivity icons were designed by Kevin Mullet while at Sun Microsystems.

REFERENCES

- [1] Allen Cypher. EAGER: Programming repetitive tasks by example. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 33–39, (New Orleans, LA, Apr. 28-May 2) 1991.
- [2] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.
- [3] Gene Fisher, Dale Busse, and David Wolber. Adding rule-based reasoning to a demonstrational interface builder. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 89–97, (Monterey, CA, Nov. 15-18) 1992.
- [4] James Foley, Won Chul Kim, Srdjan Kovacevic, and

- Kevin Murray. Defining user interfaces at a high level of abstraction. *IEEE Software*, 6(1):25–32, Jan. 1989.
- [5] Martin Frank and James Foley. Model-based user interface design by example and by interview. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 129–137, (Atlanta, GA, Nov. 3-5) 1993.
- [6] Martin Frank and James Foley. A pure reasoning engine for programming by demonstration. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 95–101, (Marina del Rey, CA, Nov. 2-4) 1994.
- [7] Thomas Kuehme and Matthias Schneider-Hufschmidt. SX/Tools - An open design environment for adaptable multimedia user interfaces. *Computer Graphics Forum*, 11(3):93–105, Sep. 1992.
- [8] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):277–304, Oct. 1993.
- [9] David Maulsby, Ian Witten, and Kenneth Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proceedings of Siggraph*, pages 127–136, (Boston, MA, Jul. 31-Aug. 4) 1989.
- [10] Brad Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
- [11] Brad Myers, Richard McDaniel, and David Kosbie. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of INTERCHI, ACM Conference on Human Factors in Computing Systems*, pages 293–300, (Amsterdam, The Netherlands, Apr. 24-29) 1993.
- [12] Brad Myers, Brad Vander Zanden, and Roger Dannenberg. Creating graphical interactive application objects by demonstration. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 95–104, (Williamsburg, VA, Nov. 13-15) 1989.
- [13] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A system for demonstrational rapid user interface development. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 167–177, (Snowbird, UT, Oct. 3-5) 1990.
- [14] Piyawadee Sukaviriya, James Foley, and Todd Griffith. A second generation user interface design environment: The model and the runtime architecture. In *Proceedings of INTERCHI, ACM Conference on Human Factors in Computing Systems*, pages 375–382, (Amsterdam, The Netherlands, Apr. 24-29) 1993.
- [15] David Wolber and Gene Fisher. A demonstrational technique for developing interfaces with dynamically created objects. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 221–230, (Hilton Head, SC, Nov. 11-13) 1991.