Model-Based User Interface Design
By Demonstration and By Interview

A Thesis
Presented to
The Academic Faculty

by

Martin Robert Frank

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
December 1995

Model-Based User Interface Design
By Demonstration and By Interview

Approved:

____
James Foley, Chairman

____
Ashok Goel

____
Scott Hudson

____
Christine Mitchell

____
Brad Myers

____
Piyawadee Sukaviriya

Date Approved________________

FOREWORD

Completing a dissertation typically requires an unprecedented level of commitment, persistence and patience from its author, and this one was no exception. First and foremost, I would like to thank my parents, my sister, and Christy Gerlach for the support that made everything possible.

Jim Foley made me realize how much better the German term “doctoral father” describes the nurturing role of an advisor.

I would also like to thank Brad Myers for his close involvement, which went far beyond the call of duty for an external committee member.

Finally, I want to thank the following individuals for volunteering their time for usability experiments and pilot studies: Gregory Abowd, Krishna Bharat, Lucy Gibson, Mark Gray, Chris Mitchell, Jayakumar Muthukumarasamy, Jim Pitkow, Spencer Rugaber, Erica Liebman Sadun, Matthias Schneider-Hufschmidt, Noi Sukaviriya, Lein Ton, and Maurizio Vitale.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graphical applications are easier to use than their character-based predecessors, but they are also harder to construct. Today, most graphical applications are built by hand-writing low-level code that makes calls to a subroutine library of user interface primitives. There is little wrong with this approach in a commercial setting. However, it presents significant problems if a non-programming audience is to participate in designing, building and modifying user interfaces.

This thesis takes a new approach towards this problem based on a special-purpose specification language and on novel demonstrational tools. In this approach, the designers first use the demonstrational tools to specify user interface behavior. As they do so, a language-based specification is generated which they can inspect. They can then experiment with editing the specification directly, using a test-drive mode to observe the effect of their changes.

The thesis contributes to the state of the art in three aspects. First, its specification language, the Elements, Events & Transitions model, is the first user-level language for interface behavior explicitly designed to be used with demonstrational tools. Second, its demonstrational tools, most notably Grizzly Bear, cover an unusually wide spectrum of user interface behavior, and are unique in keeping their reasoning independent of the characteristics of any particular user interface toolkit; we also tested them in usability experiments. Finally, the thesis is the first to explore in depth how to best combine the ease-of-use of the demonstrational approach with the expressive power of the model-based approach.

CHAPTER I

INTRODUCTION

Graphical user interfaces have made computers accessible to nearly everyone. At the same time, constructing graphical user interfaces has remained a very hard task. The most difficult part of constructing such interfaces is in describing their dynamic behavior. There has been much research aimed at reducing this difficulty.

One approach has advocated the use of high-level declarative user interface specifications (“models”) which relieve programmers from hand-coding all of the user interface functionality in a standard programming language. While this approach is powerful, it contributes little towards helping a general audience specify behavior because of the formal nature of its specification languages.

Another approach has employed machine learning techniques to infer sequencing information from demonstrations of behavior. This approach has shown great promise in letting non-programmers specify behavior. However, the range of user interfaces that can be built by demonstration is inherently limited, and having to demonstrate all of the behavior is a tedious and repetitive process.

Our focus is on combining these two approaches in a way that allows users to get started quickly by demonstrating behavior while at the same time being taught how to use the more powerful underlying language. In this introductory chapter, we will first describe the two approaches in more detail (Sections I.1 and I.2). We will then introduce our approach (Section I.3).

I.1 Model-Based User Interface Design

The discipline of model-based user interface design advocates the explicit denotation of the conceptual abstractions that underlie a user interface. This formal description can then be used to drive the user interface at run time, and it can also serve additional purposes. Humanoid [Szek92, Szek93] and UIDE [Fole89, Suka93] are typical model-based systems.

I.1.1 Strengths

The primary strength of user interface models is in their descriptive power and in their high level of abstraction. This is because they are geared towards the specification of user interface behavior, and need not cover the same breadth as general-purpose programming languages.

Another strength of user interface modelling languages is that they can additionally drive a new generation of high-level tools. For example, they enable the automatic checking for completeness, consistency, reachability and “undoability.” [1] Other tools making use of such models are user interface generators, user interface transformers and automatic help generators. (Section II.2 contains a more detailed discussion of model-based tools.)

I.1.2 Weaknesses

There are also some inherent weaknesses of the model-based approach. The most pressing problems are in the usability of model-based tools.

One problem from a usability perspective is the immediate exposure to abstraction. That is, the high level of abstraction advocated by the model-based approach also implies that the designers must understand these abstractions. This requires abstract thinking, plus a willingness to spend a significant amount of time learning the abstractions of a particular modelling language before any interface can be built. This is in contrast to more visually-oriented lower-level tools such as user interface builders which can usually be mastered in minutes.

A related problem is the need to use a formal computer language. That is, in addition to the difficulty of working at a more abstract level, the designer must also possess the ability to communicate the desired abstractions to the machine. This necessitates the use of a formal language (be the language of a textual or of a graphical nature).

---

1. A possible completeness property is that all objects that can be created can also be deleted afterwards. A consistency property may be that common sub-dialogs are identical. Reachability analysis is concerned with making sure that every part of the dialog can be reached from any other part. Finally, a proper dialog model can also enable checking for the availability of “undo” for each command.

Finally, specifying the behavior of user interfaces at a high level of abstraction can sometimes make it difficult to understand the behavior at the user interface level. For example, assume that the designer has observed that pressing a certain button has the unintended side effect of deselecting all objects. If all of the behavior is described imperatively at a low level, she can look for the procedure invoked by the button press, and then examine each of the statements there - one of them must directly or indirectly trigger this behavior. In contrast, if the behavior is specified via a higher-level model, it may take intimate knowledge of the run-time interpreter for this model to identify which part of the model causes this behavior at the user interface level.

I.2 User Interface Design by Demonstration

Over the last ten years, a new approach towards sequencing specification has emerged that lets designers describe user interface behavior through examples. That is, the designers give concrete examples of the desired behavior rather than having to deal with an abstract sequencing specification directly. Peridot [Myer88] and DEMO [Wolb91] are typical demonstrational systems for user interface design.

I.2.1 Strengths

The primary strength of programming by demonstration, and its principal motivation, is that giving examples of desired behavior takes less cognitive skill than formally specifying the same behavior.[2] Assuming that the designers are already familiar with a user interface layout tool all they need to learn is how to give examples of behavior. While this also takes skill, it compares favorably with having to learn a formal language.

Another strength of programming by demonstration is that giving interactive examples is less intimidating then textually specifying the equivalent behaviors. Visually-oriented designers in particular are more likely to be willing to give examples than they are to invest time up-front in learning a formal language.

I.2.2 Weaknesses

However, programming by demonstration also has its drawbacks. For one, there are inherent theoretical limits on its expressive power. To understand this issue, imagine that you are defining a complex mathematical function by providing a number of its points. An external observer can now draw a curve that goes through all of these points, but she can only approximate the function that you had in mind.

Or, consider that any non-trivial program cannot be exhaustively tested for correctness. Is is consequently also impossible to construct such a program from

---

2. For example, KidSim [Cyph95] combines Programming By Demonstration with a visual language to enable children as young as eight years old to interactively specify the behavior of animated characters - a task they could typically not accomplish with a conventional programming or scripting language.

a finite set of examples (because this example set would otherwise, of course, present just such an exhaustive test for correctness).

The practical limits on the expressive power of the approach are even more severe because the number of examples for defining a single behavior must be very small (less than, say, ten). Hence, assuming that the system has no prior knowledge of what will be demonstrated, it cannot infer complex behavior at all. Giving the system prior knowledge of what will be demonstrated does not solve this problem either because it is impossible to anticipate all possible demonstrations in a complex domain.

Another problem with programming by demonstration is that one cannot have complete confidence in what has been inferred without inspecting a static representation of the inferred behavior. After a demonstration, one can go into a test-drive mode in order to test the behavior with a few cases, but one can never have complete confidence in an inference without examining it in a symbolic form.

Finally, demonstrating all of the behavior for a large design can be frustratingly tedious. This is because by demonstration alone the designer cannot easily define a common behavior once and then parameterize it for future reuse.

I.3 Model-Based User Interface Design by Demonstration

This thesis explores combining the above approaches in a way that allows designers to get started quickly by demonstrating behavior, while not being later limited only to functionality that can readily be demonstrated. In the broadest terms, the challenge is to find a new approach that combines the strengths of the previous two approaches while avoiding their weaknesses. More specifically, it must be possible for the designer to start demonstrating behavior without having to know anything about the underlying modelling language beforehand. It must also be possible to later transfer knowledge acquired while demonstrating towards working with the modelling language directly.

Simply putting an existing demonstrational system on top of a separately conceived modelling language does not make for such a system. This is because the modelling language must lend itself to demonstration, and because it must not contain a large number of constructs that cannot be demonstrated, as the designer could otherwise not transfer knowledge acquired while demonstrating. Additionally, demonstrations must translate naturally into constructs of the underlying language if the designer is to understand the relationship between a demonstration and its resulting specification.

This thesis presents a new, purposefully minimal modelling language called the Elements, Events & Transitions model that supports demonstrational tools well, and facilitates the learning of its constructs by demonstration.

This thesis also presents a novel demonstrational tool called Grizzly Bear whose demonstrations translate naturally and one-to-one to the constructs of this language, and which covers a unique range of user interface behavior.

While both components contribute to their respective fields in their own right, it is their seamless collaboration that makes our design environment unique.

The structure of this thesis is as follows. Chapter II first reviews related work. Chapter III then introduces the Elements, Events & Transitions model. Chapter IV is concerned with demonstrational tools operating on top of this model. Chapter V describes the Interview Tool, an additional tool within our design environment which can sometimes generate initial models by asking a series of questions about a current interface layout. Chapter VI present the results of user tests. Chapter VII contains our evaluation of this research, and concludes with a discussion of possible extensions and future work.

CHAPTER II

PREVIOUS WORK

This chapter discusses related previous research, divided into three parts. We will first discuss user interface specification languages (Section II.1), and then review several tools that can only be made available if such a specification exists (Section II.2). We will then describe previous demonstrational tools (Section II.3).

II.1 Previous Work on User Interface Modelling

The focus of this section is on formal specification languages for describing the static content and the dynamic behavior of user interfaces. Our interest is in specification languages that operate at a higher level of abstraction than programming language code that calls subroutines of a user interface library.

MIKE [Olse86] is one of the oldest user interface management systems supporting graphical user interfaces. Its specification consists of possible end-user actions. The actions can be parameterized with predefined types like String and Point or with strings describing application-specific objects such as Resistor and Wire for a circuit design application. However, this simple model did not support defining what an application-specific object is. For example, it is not possible to state that a wire is an object which connects two other objects.

Higgens [Huds88] uses an application data model which was inspired by semantic database models. The model contains application entities and relationships. For example, musical notation can be modelled as consisting of the entities Instrument, Piece, Measure and Note. These entities have relationships to each other. For example, Measures and Notes are in a one-to-many “contains” relationship. Higgens is most notable for using this application data model at run-time, when it serves as the application interface to the user interface management system. In this way, the application need only be concerned with updating application data. It does not need to be involved in presenting data to the user.

UofA*’s application model [Sing89] consists of actions and parameters and is similar in spirit to MIKE’s model. The user can define application-specific types only in a very limited sense by specifying ranges. For example, an Angle type can be defined as “Angle=[0:360];”. From a programming language viewpoint, the designers can use predefined simple types like boolean and integer ranges, but they cannot define classes or records.

ITS [Wiec89,Wiec90] provides a layered architecture for user interface management. Its four layers consist of user interface primitives like buttons and choice boxes, a rule-based user interface generator, a dialog control component, and application routines. Its main focus is on the encoding of user interface style rules, which make use of a detailed model of the data to be presented. For example, an employee record can be declared as a field name of type String, an address of type Address and a manager of type Employee. The interface generator can then create a dialog box for display of such a field. The user interface generation process can be controlled by modifying the style rules.

HUMANOID's [Szek92,Szek93,Luo93] model consists of commands, objects, global variables and data flow constraints. Commands have associated inputs (parameters) and preconditions for their applicability. An input describes one parameter of a command by defining the type, a predicate for semantic input validation and other properties. Application objects group commands and simple objects (variables) into a semantically meaningful entity. The data flow constraints are the control element in the model, they specify the dependencies between inputs, variables and objects.

UIDE [Fole89] is a set of model-based tools. The original model consisted of actions, attributes, and an object hierarchy. Actions have parameters, pre- and postconditions. The pre- and postconditions capture part of an application action's semantics. The precondition describes when the action is available ("delete_gate is only available if there is a gate on the screen") and the postcondition specifies a modification of the state after the action executes ("after the add_gate command, the number of gates has increased by one"). The captured semantics are limited; the original UIDE model had no knowledge about where the new gate is located on the screen, for example. Attributes belong to objects and are of a fixed type like Integer or Real. Objects declare attributes and actions, they would be called "classes" in today's terminology.

The UIDE knowledge base has since been refined several times. A major driving force for the new model was its use for automatic generation of context-sensitive procedural animated help [Suka90]. This required much more detailed knowledge about the elements of an user interface, like the placement of objects on the screen. Another driving force was a new user interface generator [Kim90] which had its own requirements like logical grouping of application actions. One of the most important model additions was the “has-a” relationship (aggregation) in addition to the existing “is-a” relationship (specialization). For example, a gate of a circuit design application could be defined as consisting of several input pins, the gate body and an output pin. Another important model addition were interaction techniques. Interaction techniques are a specification of how the user invokes an action and how parameters are supplied to that action. A simple interaction technique example is pushing and releasing a mouse button over an user interface element, a more complex technique is dragging by pressing a mouse button, moving the mouse and by then releasing the button. The techniques capture user interface functionality at a much lower level than in the previous model, mainly to support the animation component.

Table 2-1 gives an overview of the specification languages presented in this section.

The User-Action Notation (UAN) [Hart90] is another well-known specification language for user interface behavior. However, it is intended for human-human communication rather than machine interpretation.[3] We will limit our discussion of previous work to machine-readable specifications here. UAN is nevertheless interesting because it could be made machine-readable by formalizing it further. However, pushing UAN too hard into this direction risks creating a language which could be too formal for human-human communication while still not being formal enough for machine interpretation.

Finally, it is worth noting that there are many related modelling languages outside the user interface domain. For example, LOOM [MacG91] is a general-purpose knowledge representation language that is widely used in the Artificial Intelligence community. It provides concepts and relations (a data model in our terminology) as well as actions and productions (a control model in our terminology). In many respects, knowledge representation languages such as LOOM are more general and powerful than the user interface-specific modelling languages presented in this section. However, their generality often incurs a cost in terms of storage overhead and run-time performance that can preclude their use in interactive software.

---

3. Citing from [Hart90], page 184: “Because UAN is in the behavioral domain, it should not be confused with, for example, specification languages for program behavior. ... Therefore, UAN is not a replacement for constructional representation techniques; it serves in a different domain.”

Table 2-1: Overview of Application Models

	Actions and Parameters	User-Defined Parameters	Action Grouping	Parameter Validation	Declarative Sequencing	Application Objects	Design Time Data Model	Run Time Data Model
Mike 1986	Yes	Only as Textual Strings	No	No	No	No	No	No
Higgens 1988	No	No	No	No	No	Yes	Yes	Yes
UofA* 1989	Yes	Ranges, Enumerations	No	No	No	No	No	No
ITS 1990	Yes	Yes	No	No	No	Data Objects, No Actions	Yes (Pre-defined Types)	No
Humanoid 1990-	Yes	Yes (Programming of Inputs)	Yes	Yes (Input Predicates)	Dataflow Constraints	Yes	Variables of Object Types	No
UIDE 1988-	Yes	Yes (Object Parameters)	No	InterParameter Constraints	Pre- and Postconditions	Yes	Variables of Object Types	No

II.2 Previous Work on Model-Driven User Interface Tools

The preceding section presented various user interface models. This section describes tools that take advantage of such a model.

Cartoonist [Suka90] automatically generates animated and context-sensitive help for the end user. The generation is based on a sequencing model that consists of user interface actions. Each of these actions has a pre-condition that must hold for the action to be applicable and a post-condition that asserts what will be true after its execution. Cartoonist can answer questions of the form “how do I enable this action” by searching for an action sequence which satisfies the precondition of the disabled action. The system then executes this sequence in slow motion by generating artificial user events, making use of knowledge about the animation of user interface techniques such as menu invocation, button press and text field fill-in. Cartoonist probably presents the visually most compelling case for declarative user interface models.

DON [Kim90] is a user interface generator that also makes use of a comprehensive declarative model. Its model includes a class hierarchy of application objects that are to be presented to the user. Each of these application object contains a list of its attributes as well as a list of applicable actions. Dependencies between actions are expressed using pre-and postconditions.[4] The high-level,

---

[4] The obvious similarity to Cartoonist’s model is not accidental as both tools have common roots in the User Interface Design Environment (Section II.1).

declarative knowledge of application-level entities enables the system to suggest an overall organization of the user interface in addition to simply suggesting the layout of individual dialog boxes.

Srdjan Kovacevic explored ways of automatically transforming user interface behavior throughout an application [Fole89,Kova92]. Consider that you have designed a user interface in which one first selects an operation and then selects the objects to which the operation is applied, and that you later realize that reversing that order makes for a better design. Representing the interactions in an appropriate high-level format allows Kovacevic’s system to automatically, painlessly and consistently change this behavior throughout the interface - a task which would otherwise require identifying and changing programming language code in many places. These transformations are especially valuable in the early design phases, where designers can quickly explore alternative interaction paradigms via transformation of a design prototype.

USAGE [Byrn94] can translate a high-level declarative user interface model into an NGOMSL model [Kier88] that is suitable for subsequent automated interface evaluation. This automated evaluation capability comes “for free” assuming that such a higher-level model has already been constructed for other purposes, such as driving the user interface at run-time.

It is worth noting that the actual modelling language used by each of the systems above differs, and that each language is naturally geared towards its specific purpose. It is currently an open research question if a *universal* modelling language exists that can support all of these tools, and that can efficiently drive the user interface at run-time as well. The Mastermind project [Szek95] currently is the most ambitious attempt of providing a comprehensive and detailed user interface model.

II.3 Previous Work on Demonstrational User Interface Tools

A number of demonstrational user interface tools have been built during the last ten years. We will first describe relevant systems individually in chronological order, and then provide a summary of their key characteristics.

Peridot [Myer88] supports designing scrollbars, buttons, choice boxes and similar objects by demonstration. It was the first user interface tool to provide for the interactive specification of *behavior* in addition to layout.

The primitives of Peridot's inference mechanism are rectangles, circles, text lines and icons. Peridot uses active values to describe the state of an interface. Active values are global variables which maintain a list of objects to notify when their value changes. For example, a choice box is associated with an active value that enumerates the choices. Both the application and the end user can change this value and the other party will be informed of the change. Another example in Peridot is the mouse pointing device, which is attached to an active value consisting of a cartesian coordinate and three booleans for the mouse buttons.

Peridot created the research field of “by-demonstration” tools. The major drawback was its heuristic, rule-based reasoning which in essence codifies educated guesses of what the user is trying to do. This methodology works well if the guesses are correct but it fails fatally if they are not because it does not present an escape mechanism. Worse, the user is then always left to wonder if the system could make the right guess when supplied with a slightly different demonstration, or if it just cannot draw the desired inference at all. We encountered the same limitations with our rule-based tool, the “interview component” (Chapter V).

Lapidary [Myer89] focused on creating application-specific objects. It also used constraints but replaced Peridot’s active values with interactors [Myer90a] as its way of handling user input. To specify a constraint by demonstration, the user first selects the user interface event which triggers the constraint, such as object selection with the mouse. The designer initiates two system snapshots, before and after the change to the object (for example to make its text italic). The system then creates a constraint which makes the text of highlighted objects italic.

Metamouse [Maul89] learns graphical procedures by example. The user first invokes a special teaching mode. Metamouse then watches the user perform graphical editing operations and uses generalization to identify the steps, loops, and branches in this procedure. Metamouse used the proximity of objects to reduce the amount of computation required for its inferencing. “Metamouse is near-sighted but touch sensitive. The user understands that relations at a distance must be constructed, for example by using a line to demonstrate alignment" [Maul89, page 128, end of first paragraph].[5]

Druid [Sing90] lets users attach simple functionality such as enabling, disabling, hiding and showing buttons. It is a user interface management system with demonstrational capabilities. It differs from Peridot and Lapidary in that the system is watching the designer perform interactions over time rather than taking snapshots of the state and reasoning about the difference in the states. Druid's approach is somewhat similar to macro recording in Emacs, FrameMaker or other text processing systems. Druid lacks a static representation of the recorded behavior, so that the only way to see what has been recorded is to replay it and the only way to edit is to re-record. The "recording" approach works best when no parameters are involved in the interaction. However, Druid does have some capability to deal with arguments. For example, the designer can demonstrate that the arguments to a "create rectangle" function are two mouse clicks which specify

---

5. It is interesting to compare this search space reduction technique to the one that we have used for our demonstrational tools (Chapter IV). Grizzly Bear could accordingly be described as "motion sensitive" rather than "touch sensitive". There is no need for auxiliary objects as in Metamouse, but the user has to understand that an object must be moved (or otherwise changed) for it to become relevant to a demonstration. Both strategies aim at reducing the number of objects that must be considered in the generalization process.

opposing corners of a rectangle. However, it is generally not possible to demonstrate a parameterized interaction with a single demonstration. In conclusion, the macro recording approach works best for parameterless interactions and quickly becomes arkward for more complex interactions.

Eager [Cyph91] watches users perform operations and detects and automates repetition. Eager differs significantly from the other demonstrational systems discussed in this section. It does not synthesize a program but rather just automates repetition. Eager does not have to be explicitly invoked, it is rather constantly running in the background. When it detects repetition it displays its logo in the menu choice that it anticipates will be selected next. The users thus get a subtle, unobtrusive hint that they can use Eager to automate repetition. Eager presented a usability breakthrough for by-demonstration systems because it did not require any special skills from its users.[6]

DEMO [Wolb91] uses a stimulus-response paradigm for demonstrating the behavior of graphical objects. The designer specifies a triggering event, the stimulus, and then demonstrates the intended behavior, the response. This paradigm

---

6. However, it should of course again be noted that Eager only addresses the automation of user actions (the users could accomplish their tasks without Eager). Most other systems presented in this section address the much harder task of empowering them to define user interface behavior, a task they could otherwise not accomplish at all.

supplies a natural mental model of how the demonstrational system works. DEMO does not use predefined widgets such as buttons and sliders but drawing primitives such as lines, circles and rectangles. The possible stimuli are atomic mouse events such as mouse button presses and releases. For example, it is possible to specify that a line at a fixed position should appear when the left mouse button is pressed by specifying the mouse press as the stimulus and then drawing the line in response, and finally specifying that the stimulus should execute the response. It is also possible to specify that a parameterized line should be created depending on where the mouse was clicked and where it was released, this is done exactly as before but by stating that the stimulus should initiate the response. DEMO does some linear generalization similar to Peridot, so that it is possible to link an angle of a gauge to a numeric text field.

DEMO II [Fish92] is a successor of the DEMO system which has been augmented by a rule base. Some of the rules try to automatically recognize the stimulus for a demonstrated behavior. Other rules try to infer non-linear relationships. Inferring non-linear relationships is an important goal but it is inherently limited because complex relationships would require large numbers of examples. DEMO II uses auxiliary objects such as invisible lines and ellipses to specify the constraints between geometric objects. DEMO II is the first by-demonstration system that tries to overcome the linear-relationship barrier but it succeeds only in a limited domain. It is also notable for incorporating both rule-based heuristics and more mathematically-thorough inferencing. Heuristics are appropriate for capturing commonly used behavior but there must be a fall-back mechanism. We have also used a dual approach: The Interview Tool of Chapter V is our heuristic component, while the demonstrational tools of Chapter IV use mathematically-thorough reasoning. In a nutshell, heuristic components can be more forgiving while the mathematics-based components can make more general inferences.

Chimera [Kurl93] infers constraints between graphical objects given multiple snapshots[7]. For example, it can infer that buttons should be distributed evenly when their window is resized. It can also infer constraints between objects in an application's main window, such as the components of a two-dimensional projection of a lamp with a flexible metal arm (a “Luxo” lamp). It is maybe the computationally-strongest interactive graphical constraint solver to date.

The user interface to the constraint solver shares some principles with our approach. One similarity is that constraint enforcement can be turned on and off. When the constraints are turned off, the scene can be directly manipulated in order to provide a new snapshot which is similar to our build mode. When the constraints are turned on, the users can see if the scene behaves in the way they intended which is similar to our run mode[8]. Another similarity is that the users do not have to have a mental model of how the system will make their demonstrated behavior work. In Chimera, the user providing snapshots of desired configurations

---

[7] The constraint solver is only part of the functionality of the Chimera editor, but we will not distinguish between the two here.

does not have to know about the concept of constraints. This is an advantage of snapshot-based inference over graphical programming because graphical programming can only free their user from some of the syntax of the domain but not from the requirement to know about the concepts[9].

Chimera cannot automatically infer parameters from given snapshots because the number of parameter candidates in a complex scene is large. An extension to Chimera lets users provide a set of integer values with each snapshot. The given values are treated in the same way as the distances, slopes and angles of the snapshot so that the provided values are constrained in the same way. For example, slider, dial and gauge widgets can be constructed in this way, similar to Peridot. The primitives of Chimera's inferencing mechanism are lines and other elements of a simple drawing editor. Chimera does not reason about objects that are dynamically instantiated and deleted at run-time.

Finally, Marquise [Myer93] uses domain knowledge to support building graphical editors. It is a by-demonstration system for creating MacDraw-style graphical editors. It contains built-in knowledge about editor-specific behaviors

---

8. This similarity is not surprising as all graphical snapshot inference mechanisms must provide a mode in which the inferred representations are not executed because the user could not arrange the initial snapshots otherwise.

9. Scripting languages require their users to know both the syntax and the semantics before they can start a new design.

such as selection from a palette and grouping of objects. It is similar in capability to DEMO II which can also handle creating lines and constraining lines to boxes. It goes beyond DEMO II in its capability to demonstrate feedback such as rubber-banding lines.

Marquise uses several mouse icons to statically represent the demonstrated dynamic behavior. For example, consider that the user demonstrates how to create a line by selecting the line mode from a palette, pressing the left mouse button in the main window, dragging and releasing the mouse button. Marquise can then show the mouse events on the screen using these icons. One of the advantages is that the user can now draw a dotted line between the mouse down and up events to indicate the kind of semantic feedback that is intended. More important, this provides for an editable static representation of the demonstrated behavior, a prerequisite for the practical use of by-demonstration systems. Marquise actually records events rather than isolated snapshots, similar to Druid, so that it can also be characterized as a generalized macro recording approach.

All of the systems discussed above use by-demonstration techniques but they are not easily compared because they have different goals and use different techniques. Nevertheless, we make an attempt to classify them in Table 2-2.

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 advantage of constant watching is that the users do not have to learn anything new to take advantage of the system. Some systems query

Table 2-2: Overview of Demonstrational Systems

	Interface	*Interface*	*Capability*	*Capability*	*Internals*	*Internals*	*Internals*	*Internals*
	Is Eager (Constantly Watches User)	Uses A Clarification Dialog	Dynamic Object Creation & Deletion	Subjective Strength in Geometric Relations	Search Space Reduction	Is Rule-Based	Temporary Behavior Storage	Output
Peridot 1987	Yes (Query)	Yes	No	Low	None[a]	Yes	Snapshots	Constraints & Behavior
Lapidary 1989	No	Yes	No	Low	None	No	Snapshots	Constraints & Interactors
Metamouse 1989	Yes (Prediction)	Yes	No	Medium	Explicit[b] (Auxil. Objects)	No	not applicable	Graphical Procedure
Druid[c] 1990	No	Yes	No	None	None	No	Event Recording	Script
Eager 1991	Yes (Prediction)	No	not applicable	not applicable	not applicable	No	Event Recording	Macro
DEMO 1991	No	Yes	Yes	High	Explicit (Auxil. Objects)	Yes (DEMO II)	Compressed Snapshots	Response Description
Chimera 1991	No	No	No	High	Explicit[d] (Auxil. Objects)	No	Snapshots	Two-Way Constraints
Marquise 1993	No	Optional[e]	Yes	Medium	None	Yes	Event Recording	LISP Code

[a] "None" in this column should be interpreted as "none published".

[b] Metamouse additionally reduces the search space by only considering touch relations.

[c] Note that only Druid's demonstrational component is discussed here, not its rule-based design assistant.

[d] Chimera also uses a variety of other techniques to reduce the solution cost of a demonstration.

[e] A feedback window is displayed; the designer can change aspects of the behavior but does not have to.

the user when they make an inference (marked with “Query” in the table), others indicate their inferences by subtly displaying their prediction of what the user is doing next (“Prediction”). The second column describes if the system asks the user for confirmation and clarification after each inference. Any inferencing system will sometimes guess wrong - the clarification dialog gives the user an opportunity to correct and fine-tune inferred behavior. The disadvantage here 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 systems. 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. We labelled systems which can detect simple relationships such as centering and touching “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 if the system reduces the number of objects that it checks for relationships. For example, some systems use auxiliary objects such as guide wires to let the user specify the relevant objects and their relationship. 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 commonly used behavior. The disadvantage is that the rules can sometimes miss even simple relationships while algorithm-based systems can handle all relationships within a certain class. The seventh column describes how previous demonstrations are captured. There are two main approaches towards capturing demonstrations. Event-recording stores the events during a demonstration by the user. Snapshot-taking records a series of states. The last column describes the output of the inferencing process.

CHAPTER III

THE ELEMENTS, EVENTS & TRANSITIONS MODEL

This chapter first motivates and introduces a model for describing user interface behavior. It then discusses advanced features of the model, and presents a computer-readable language for specifying models.

III.1 Motivation

The Elements, Events & Transition (EET) language is a special-purpose language for the design of interactive graphical user interfaces.

III.1.1 General Motivation for Special-Purpose UI Specification Languages

But is a special-purpose language a good idea at all? Many professional programmers resent having to learn “yet another language”. They have already invested considerable time in learning a general-purpose programming language - so why invest even more time in learning another language which has no more overall expressive power, and likely less? And there will normally also be a run-time cost because these special-purpose languages tend to be interpreted!

Mark Linton [Lint89, page 20] provides a good summary of these criticisms. For example, he writes: “The special-purpose language used ... is likely to be unfamiliar to programmers and user interface designers alike. Moreover, the language is usually inferior to established general-purpose languages, the debugging tools are primitive or nonexistent, and run-time overhead associated with interpreting the specification often degrades performance compared to conventional implementations."

These objections are valid. There are indeed situations where hand-writing user interface code is preferable to any other method of construction. This is especially true for commercial products that are widely distributed. The effort spent for hand-writing user interface code for such products is of virtually no concern because of the economy of scale involved. For example, a large software manufacturer can afford to employ many professional programmers which carefully hand-craft user interface code.

However, the vast majority of computer users are not professional programmers. Requiring extensive experience with a general-purpose programming language would permanently exclude them from constructing even simple user interfaces. It would also exclude them from modifying existing user interfaces to suit their needs and taste (beyond choosing from a finite number of pre-defined options). Therefore, there clearly is a need for user interface languages which are more accessible.

III.1.2 Motivation for the EET Language

The primary design rationale of the EET language is to facilitate its production with demonstrational tools. To the best of our knowledge, the EET language is the first language for describing dynamic user interface behavior with this explicit design goal.

Other important requirements are that the language is nevertheless also readable by humans (so that they can inspect and modify the output of the demonstrational tools), and that it can be interpreted efficiently.

III.2 Introduction to the EET Model

The Elements, Events & Transitions (EET) model serves two purposes. First, it is used as the underlying model for demonstrational tools. Second, the EET language[10] doubles as a user-level scripting language when the interactive tools are inadequate, or when it is easier or faster to specify the behavior directly.

In response to these requirements, the Elements, Events & Transitions model introduces only the three abstractions that give the model its name. This minimality facilitates its construction by demonstrational components and does not unnecessarily burden novices in learning the language. In addition, strictly separating static properties (Elements) from dynamic properties (Transitions) supports our design methodology of specifying static properties first (by direct manipulation) and dynamic properties later (by demonstration).

---

[10] The EET *model* presents the abstractions available for describing user interface behavior. The EET *language* is a particular syntax for specifying such models.

III.2.1 Elements

All graphical user interfaces are at least conceptually composed of elements. For example, most interfaces use button elements, slider elements, window elements and so forth. Thus, virtually all modern toolkits, or subroutine libraries, use programming language abstractions to describe those components. These abstractions are often called “objects.” We do not use this term because it is already over-used. Our abstraction for describing these user interface components is called an *element*.

It is worth keeping in mind that a user of our interactive tools does not have to be exposed to a textual specification for elements. This is achieved by editing the element space by direct manipulation, via a conventional user interface builder. Users can add, remove and modify elements in this way. Nevertheless, we will introduce the textual syntax for specifying elements in this section as this will help us make our discussion more concise. More advanced users may also want to introduce abstract elements (elements that have no representation within the user interface) which can only be edited textually.

An element is a collection of *attributes*. An attribute typically consists of a type, a name, and a value. For example, the attribute below describes a numeric attribute called “width” and gives it a value.

```
<integer> width     80;
```

Attribute values can be quoted. This is necessary only if the values contain blank spaces such as in the example below.

```
<string>  label     "Press Me"
```

The type of an attribute is optional. For example, we could earlier have also expressed the attribute as shown below.

```
width     80;
```

Omitting the type of an attribute implies that its values will be stored as strings and that its type will be marked as “unknown”. Not specifying a type can be convenient. However, it has the drawback that access and storage of that attribute will not be as efficient as they could be. As we will see in Chapter IV, it also implies that the demonstrational components will not be able to draw type-specific inferences.

It is also legal to specify type names that do not exist internally, say “color” or “font”, with the same drawbacks (the built-in types are “string”, “integer”, and “boolean”). However, this is still preferable to not specifying types at all because the EET interpreter can then warn about assigning values of attributes with differently named types.

Attributes of user interface elements nearly always have exactly one value. However, attributes can also be multi-valued. In the user interface realm, this can for example be used to maintain a list of currently selected objects.

```
<string>  selected  gate2 gate5 gate7;
```

The fact that attributes may have multiple values implies that attributes may have no value at all. In the above example, the *selected* attribute may have no value (there may not be any objects selected).

```
<string>  selected  ;
```

Note that by convention the type specification of multi-valued attributes uses the singular of the type of its values (“string”, not “strings”).

Now that we have defined what attributes are, we can simply define *elements* as a collection of attributes. For example, a button can be described by a name, a position, a size, and so on, as shown below.

```
Element {
     <string>     class      Button;
     <string>     name       b;
     <integer>    x          220
     <integer>    y          140;
     <integer>    height     25;
     <integer>    width      80;
     <boolean>    enabled    true;
     ...
}
```

Attribute names must be unique within an element. For example, it would be illegal to specify two attributes with the name “height”.

Elements will often be given a name, which is expressed by a string-valued attribute called “name”, *b* in the above example They can then be referred to by this name. Elements do not have to have names because they can also be referred to by the values of their other attributes.

It is interesting to note that the “name” attribute is not special in any way other than providing a name for the element containing it. That is, its value can be computed like the value of any other attribute, and the value of a name attribute can be used to compute others.

Attribute values are atomic - they cannot be composed of other values. That is, a value can be of type *string* or *color*, but it cannot be of type *element* - there is no nesting of elements. However, the same effect can be achieved by referring to elements in attributes. The *selected* attribute is an example of this technique. While the values of this attribute are simple names they can be de-referenced to access the content of the named elements.

III.2.2 Events

The second abstraction of the Elements, Events & Transitions model is the *event*. Just as their real-world counterparts, EET events signal that something significant has happened.

The EET model does not make assumptions about the type of events that can occur. The model only provides structure to those events, and gives the user a way of specifying what should happen in response to them.[11]

We will again introduce a textual notation for events while reminding the reader that beginning users of our interactive tools only need to know about the

---

[11] A small piece of software situated between the window system and the EET interpreter establishes the actual types of events available.

concept of an event ("when I press the delete button, I want the currently selected objects to go away"), not any syntax. As before, presenting the textual notation here will make our discussion more concise.

An EET event consists of the name of the object on which the event occurred, the name of the event, and a list of event parameters. The simple event below expresses that a "delete" button was pressed.

```
deleteButton.pressed()
```

In the example of a button, we will normally not be interested in potential parameters of the event, such as the exact position of where the click occurred on the button. However, there are also many situations where we are interested in parameters. For example, we will normally be interested in where the click occurred on a canvas object.

```
canvas.pressed(
          <integer>   x    213,
          <integer>   y    79)
```

There is some similarity between an element attribute and an event parameter in that they both have a name and a - potentially unknown - type. In contrast to attributes, event parameters always carry exactly one value. Just as for attributes, the order of event parameters is of no significance because event parameters are always referred to by name. That is, the event below is indistinguishable from the one above.

```
canvas.pressed(
          <integer>   y    79,
          <integer>   x    213)
```

Just as for attributes within an element, the same parameter name cannot occur twice within an event.

III.2.3 Transitions

Transitions are the last abstraction of the Elements, Events & Transitions Model. They describe how the elements change in response to events.

Simple transitions often translate one-to-one to an informal natural-language description, such as “the Properties window should disappear if the user presses the Cancel button.”

Transitions formalize this natural-language description to make it machine-executable. As stated above, we will introduce the textual syntax both to make our discussion more precise and to show which language constructs are available for advanced users. There is no need to know this textual syntax to start building a functional user interface.

Transitions consist of two parts. The *transition header* describes under which circumstances the transition is invoked. The *transition body* describes how EET elements change once it is invoked. The transition below gives a simple example.

```
Transition propertiesCancelButton.pressed()
{
          propertiesWindow.visible := "false";
}
```

In this example, the header describes that this transition is invoked if the user causes a “pressed” event over an interface element called *propertiesCancelButton*. The body describes that the *propertiesWindow* becomes invisible in response.

A transition body consists of a series of *statements*. The body of the above transition contains the most common statement, the assignment statement. It sets the value of an element attribute to the value provided on its right-hand side.

Assignment statements are not restricted to setting values to constants, they can also contain *value expressions* on their right-hand sides such as the ones below. (The assignments state that an element named *e* doubles in width while its center remains at the same location.)

```
e.x := e.x - 1/2 * e.width;
e.width := 2 * e.width;
```

Assignment statements always specify an attribute reference on their left-hand side. Their right-hand side, the value expression, can refer to constants, attributes, and event parameters. The transition below refers to an event parameter. (The transition specifies that mouse movement events on element *e* will cause it to follow the mouse.)

```
Transition e.mouseMoved(integer x, integer y)
{
          e.x := x;
          e.y := y;
}
```

As discussed earlier, element attributes can be one-valued or multi-valued. Assignment statements deal with one-valued attributes.

There are also several statements for changing multi-valued attributes which maintain a list of values. Among them are statements for clearing a list, for adding a value to a list, and for removing values from a list. Below is an example for clearing the values of the *selected* attribute of the *bb* element.

```
bb.selected := ;
```

Below are two examples of adding values to the end of a list. The right-hand side does not have to be constant, it can be an arbitrarily complex value expression just as in regular assignment statements.

```
bb.selected += pin231;
e.someNumberSequence += 4 * 8 * 66;
```

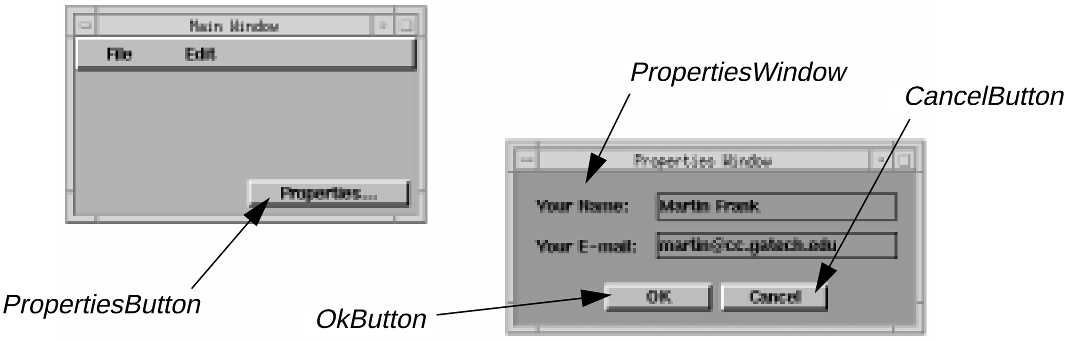
The following statement will delete the first matching value from a list. The right-hand side can again contain a value expression.

```
bb.selected -= "element to be deleted";
```

The above three statements manipulate list items by their value, and have proven sufficient for our purposes. Additional statements that can manipulate list items by their position may need to be added for more list-intensive models (e.g. “add this new element after the third existing element” or “remove the first element”).

III.2.4 Example: Showing and Hiding an Auxiliary Window

We will now give a minimal example of a user interface driven by an Elements, Events & Transitions model. Figure 3-1 shows the windows of this example interface. The user interface elements are represented as EET elements. The figure shows the relevant element names in italics, pointing to their respective elements with arrows. (Neither the italic labels nor the arrows are part of the user interface.)



Your Name: Martin Frank
Your E-mail: martin@cc.gatech.edu

Figure 3-1: Showing and Hiding an Auxiliary Window

The functionality we want to describe in this example is that the Properties window is made visible by pressing the Properties button. The window becomes invisible again when the user presses the Cancel button or the OK button. The Properties button is disabled when the Properties window is already visible. An Elements, Events & Transitions model for this functionality is shown below.[12]

```
Transition PropertiesButton.pressed()
{
     PropertiesWindow.mapped := 1;
     PropertiesButton.textForeground := "white";
}
Transition CancelButton.pressed()
{
     PropertiesWindow.mapped := 0;
     PropertiesButton.textForeground := "black";
}
Transition OkButton.pressed()
{
     PropertiesWindow.mapped := 0;
     PropertiesButton.textForeground := "black";
}
```

All concrete EET models depend on the characteristics of the window system and user interface toolkit that they are attached to.[13] In this case, the toolkit provides an integer-valued attribute called *mapped* that controls the visibility of elements.[14]

We will later discuss ways of expressing the same model in a different, less repetitive way. For example, we will see how we can tie the text color of the Properties button directly to the visibility status of the Properties window, so that we do not have to explicitly set the text color of this button whenever we change the visibility of the window (Section III.3.6). We will also see how we can avoid the duplication of EET statements between the OK button and the Cancel button (Section III.3.7). In the chapter on our demonstrational tools, we will also show

---

12. Note that in our prototype implementation of the EET model all names are global (such as “OKButton”). An improved version could use hierarchical names to avoid name-space pollution (e.g. “/PropertiesWindow/OKButton”).

how this model can be interactively generated from demonstrations (Section IV.1.2).

The EET model presented above obviously represents only a small segment of the overall functionality. Later in this chapter, we will see how we can express more complex functionality in EET models, and how we can attach application code (Section III.3.8).

III.3 Advanced Concepts of the EET Model

The previous section has introduced the Elements, Events & Transitions Model. This section describes its more advanced concepts. Advanced users will be able to write more expressive and powerful models if they are familiar with these concepts.

---

13. An earlier example used a string-valued attribute called “visible” for the same purpose, coming from a hypothetical user interface toolkit (Section III.2.3). The EET model does not make assumptions about the nature of the attributes, and has no attribute names hard-wired into its language. It instead relies on a translator to fill in the elements from the toolkit at run-time, and to propagate changes to the elements back to the actual toolkit. The toolkit we used for our prototype implementation (SX/Tools [Kueh92]) uses an integer-valued attribute called “mapped” to control visibility.

III.3.1 Statements that Operate on Sets of Elements

The transitions shown in the introductory section use statements that operate on one element at a time. It is also possible to use statements which specify operations on a *set* of elements at a time.

Many modern user interfaces let users select multiple objects, and then let them issue commands that affect all selected objects at once. The following statement changes the width of all selected objects simultaneously. (It is a fragment of an interaction which lets users textually specify precise properties of graphical elements.)

```
(*.selected=="true").width := numericField.value;
```

We call the parenthesized expression a *set expression*. Set expressions contain comparisons, such as the one above. Comparisons can then be combined using the boolean *and*, *or* and *not* operators (their syntax follows the C programming language conventions: “&&”, “||” and “!”).

The left-hand side of a comparison usually contains a *wildcard reference* to an attribute name. The right-hand side of a comparison consists of an arbitrary

---

14. A value of zero means that it is hidden, all other values imply that it is visible. This particular toolkit does not provide an enabled/disabled attribute for buttons so that we use their text color to communicate disabling to the end user. (Enabled buttons show black text, disabled buttons show white text.)

value expression (as defined in Section III.2.3). The following is a list of more complex set expression examples.

```
(! (*.color=="red" || *.color=="blue"))
(*.x>150 && *.x<250 && *.parent=="canvas")
(*.width < e.width + w)
```

The first expression selects all elements which do not have a color attribute whose value is red or blue. The second example selects all elements with an “x” value between 150 and 250 whose “parent” attribute contains “canvas”. The last example shows how to use a value expression in a comparison: *e.width* refers to an attribute, *w* refers to an event parameter.

The left-hand side of a comparison can not only contain attribute references but also event parameter references, list length operators and references to attributes of concrete elements. We refer to such comparisons as *concrete comparisons* as opposed to the *wildcard comparisons* above. But what then is the semantics of the following set expression?

```
Transition e.pressed(enum b)
{
          (b==1).color := "brown";
}
```

A concrete comparison which yields true means “all elements”, a concrete comparison that evaluates to false accordingly means “no elements”. Hence, if the *b* parameter in the above transition is indeed equal to one, the assignment statement will change the “color” attribute of all elements having such an attribute. If the *b* parameter is not one, nothing will happen.

This example is admittedly contrived, as set expressions containing no wildcard comparisons at all will rarely be useful in practice. However, they are useful for making standard set expressions conditional. For example, the statement below will enable a certain button only if there are currently two selected pins.

```
(*.name=="ConnectButton" &&
 bb.selectedPins==2).enabled := "true";
```

The wildcard comparison on the left limits the elements under consideration to one, namely to the one called *ConnectButton*. The concrete comparison then limits the elements to either none or all elements depending on the value of the *selectedPins* attribute of the *bb* element. In combination, the only possible element whose *enabled* attribute may be affected is the *ConnectButton* element, depending on *bb.selectedPins*.

We conclude by noting that the attribute specification *element.attribute* - which was used to introduce transitions in Section III.2.3 - is just a shorthand notation for the set expression *(*.name==”element”).attribute*. These two notations are otherwise equivalent.

III.3.2 Transitions for Sets of Elements

All the transition invocations in the introduction to the EET model were tied to events on a single object. For example, they have tied a transition to a “press” event on the “delete” button. There are certain situations where it is desirable to specify a transition that should occur *for all objects of a certain group*.

For example, assume you have built a custom “selection box” containing three buttons. Pushing any of those buttons highlights this button and de-highlights the others. (Many widget sets already provide such a selection box, but it can sometimes still be desirable to built your own, for example to be able to arbitrarily lay out the individual buttons.) One way to implement such a selection box is to provide three transitions, one for each button.

```
Transition button1.pressed()
{
     button1.reverseVideo := "true";
     // statements omitted that de-highlight others
}

Transition button2.pressed()
{
     button2.reverseVideo := "true";
     // ...
}
Transition button3.pressed()
{
     button3.reverseVideo := "true";
     // ...
}
```

However, this is repetitive and tedious so that it is desirable to be able to say “if *any* of these three buttons is pressed then...”.

In the above example, being able to specify transitions on a group of elements is merely convenient. There are also situations were there is no alternative to being able to do so. For example, consider an interface which lets users create and delete objects dynamically. In such a case, you *have* to be able to say “if *any* of those objects are pressed...” because at run-time there can be an infinite number of objects whose names cannot be known beforehand.

Transitions on a group of elements are specified by replacing the name of a concrete element with a set expression. Set expressions in transition headers use the same syntax as the set expressions in statements introduced in the previous section.

Below is an example transition that is invoked whenever a “press” event occurs on an element which has an “isSelectable” attribute that has the value “true”.

```
Transition (*.isSelectable=="true").pressed()
{...}
```

In this example, we may want to highlight the selected object by drawing a rectangle around it or by changing its color to red. But we cannot know the name of an object that we have described only by the value of its attributes. How then do we refer to the element on which an event has occurred inside such a transition?

The answer is that the EET model provides a special pseudo object reference, called *self*, that is synonymous to “the object that has just matched the set expression of this transition”. We can thus fill in the body of the above transition.

```
Transition (*.isSelectable=="true").pressed()
{
     self.color := "red";
}
```

In the case of multiple elements matching the expression, the body of the transition is executed once for every matching element so that “self” does not necessarily imply that only one element is affected.

As already mentioned in the preceding section, we can also make set expressions dependent on other properties of the current state. For example, we can make a transition for a button that connects pins dependent on there being exactly two currently selected pins (assuming, of course, that the *selectedPins* attribute of the *blackboard* element indeed keeps track of the number of selected pins).

```
Transition ((*.name=="connectButton") &&
            (blackboard.selectedPins==2)).pressed()
{...}
```

In summary, set expressions are the concept in the Elements, Events & Transitions model that allow users to specify both conditionality and simple iteration without having to introduce explicit “if-then-else” and “for-all” statements.[15]

Finally, let us introduce an alternative notation for parameter constraints that is functionally equivalent to set expressions but often more readable. The following set expression makes the execution of a transition dependent on the *b* parameter being one.[16] The transition triggers upon a *pressed* event on an element called *e*.

```
Transition (*.name=="e" && b==1).pressed(enum b)
{...}
```

The advantage of this notation is that it is consistent with the notation used for statements operating on sets of elements, and that comparisons of event parameters and comparisons of attribute values are treated uniformly. The following alternative notations are equivalent.

```
Transition (*.name=="e").pressed(enum b==1)
{...}

Transition e.pressed(enum b==1)
{...}
```

Grouping parameter restrictions with the parameter definitions in this way is often more readable than putting the parameter restrictions into the set expressions as above.

---

15. The EET model limits iteration to the form in which the body of a loop statement is conceptually executed concurrently for all affected elements. That is, the EET model does not support iteration in which the body of a loop depends on the computational results of previous iterations. The EET model does also not support recursive invocation of transitions. These advanced constructs are rarely necessary to build graphical user interface front-end functionality, and would complicate the EET interpreter. Instead, the EET model provides for the integration of custom programming-language code at any point of EET execution (as described in detail in Section III.3.8).

III.3.3 Transitions that Create and Delete Elements

The previous sections explain in detail how transition statements change attribute values of elements. This section will introduce two statements which create and delete elements.

In the Elements, Events & Transitions model, an element is never created from scratch at run-time; it is rather copied from an existing element.[17] That is, the other element serves as a prototype, and every element can be used as a prototype for others. The newly created object is then no longer linked to its prototype. That is, changing attributes of an element has no effect on the elements that were created from it earlier.[18]

Copying an element implies that the new element will contain the exact same attributes as its prototype, with the sole exception that its name attribute will differ (which will be given an artificially generated name).[19]

---

16. The example is taken from a user interface front-end that supplies an enumerated value for each mouse-down event that encodes which mouse button was pressed. In this encoding scheme the left-most mouse button is button one, the second-from-left mouse button is button two, and so on.

17. This avoids having to introduce the designer to the concept of a “class”. In addition, this makes the run-time creation of an object conceptually not different from dragging an object into the design from a palette.

The following *create statement* will instantiate a new element that is copied from an existing element called “pin”.

```
element n := create pin;
```

In this statement, *element* and *create* are keywords of the EET language. The element reference pin specifies the element to be copied. The name following the element keyword, *n*, defines a pseudo element reference that can be used to refer to the newly created object. This is necessary because we cannot know the name of a newly created object at this point. The use of pseudo element references for newly created elements is analogous to the use of the pseudo element reference *self* introduced earlier. In both cases, the use of these references is only valid within the current transition.[20]

It is possible to use a set expression instead of an absolute name in specifying the prototype of a create statement. For example, imagine that the prototype

---

18. The rationale for this decision was to ensure that the execution of a transition remains the only means of changing an element. Otherwise, a change to an element may be caused indirectly by a change to its seemingly unrelated prototype. This is not to say that run-time inheritance may not be useful in some cases, or that it could not be incorporated into the EET model as a future extension.

19. This is not different from other prototype-instance schemes like e.g. the ones used in SX/Tools [Kueh92], SUIT [Paus91], and Garnet [Myer90b].

from which a new element is copied can change at run time, and that it is marked with an *iAmTheCurrentPrototype* attribute.

```
element e := create (*.iAmTheCurrentPrototype=="true");
```

Note that a set expression in a create statement must yield exactly one element. If it does not, the EET interpreter will not perform any action other than issuing a warning.

The *delete statement* is complementary to the create statement. The following example deletes all selected pins (assuming, of course, that pins have an *isAPin* attribute and that selection is modelled with a *selected* attribute).

```
delete ((*.role=="pin") && (*.selected=="true"));
```

The statement will delete all matching elements. It is legal for its expression not to match any element, the delete statement will then have no effect.

---

20. It is worth mentioning that this is the only part of the language that makes use of a “local variable” concept, where it seems like the most natural solution to the problem of referring to newly created elements (whose automatically generated names cannot be known beforehand). We feel that the general introduction of local variables would only add complexity to the language without adding power, as it is always possible to achieve the same effect by using attributes of abstract elements as local variables (Section III.3.5).

III.3.4 Multiple Indirection in Attribute References

The left-hand side of attribute-changing statements consists of *attribute references*. Section III.2.3 introduced the simplest attribute references, of the form shown below.

```
button7.width := ...
```

Section III.3.1 then introduced attribute references that refer to a set of element attributes at a time. Below is an example.

```
(*.selected=="false").width := ...
```

We complete our discussion of attribute references by introducing attribute references that use multiple indirection. For example, the following statement changes an attribute of the layout parent of a button (assuming, of course, that the button has a *parent* attribute that indeed provides the name of its layout parent).

```
CancelButton.parent.visible := "false";
```

It is possible to combine the use of a set expression and of multiple indirection.

```
(*.selected=="true").from.color := "red";
```

All *attribute* references in an EET model are absolute. In the above example, both the intermediate and the terminal attribute references (“from” and “color”) are absolute attribute names. The concept of expressions on attributes in addition to expressions on elements existed in an earlier language design.

```
# Not a valid EET statement!
(*.selected=="true").("red") := "blue";
```

For example, the above statement used to express “set all attributes with a value of ‘red’ to the value ‘blue’ for the currently selected elements”. However, we found these attribute expressions to be of little practical value, and they were also particularly hard to read. It is of course possible to again extend the language with attribute expressions if a new application domain for the EET language should call for this additional pattern matching capability.

III.3.5 Abstract Elements and Prototype Elements

User interface elements are “real” in the sense that the user can view them and manipulate them. There are several reasons why you may additionally want to introduce *abstract elements* into an Elements, Events & Transition model.

Abstract elements should not be confused with the concept of “abstract superclasses” found in some object-oriented programming languages. An abstract EET element differs from an ordinary EET element only in not having a visual representation.

The most common use of an abstract element is as a placeholder for various variables that have no immediate visual representation. Abstract elements must be specified textually (as they per definition have no visual representation), and appear at the beginning of an EET model, before the transition definitions. A simple example is shown below.

```
Element {
          <string>  name           bb;
          <string>  selectedPins   ;
}
```

In this example, the element is used to keep a list of the names of the currently selected pins. The name *bb* is a shorthand for “blackboard”, a name which we by convention often use for an element that holds global variables (abstract elements can be arbitrarily named). Abstract elements can also be defined by the following more compact “convenience notation”.

```
Object bb {
          <string>  selectedPins   ;
}
```

There is no semantic difference between this definition and the one above it, the EET interpreter will in both cases generate the same abstract element when the specification is read in.

We have stated earlier that any element can serve as a prototype. If an abstract element is defined for the sole purpose of serving as a prototype for other abstract elements to be created, it can be distinguished using the *prototype* keyword.

```
Prototype p {
          ...
}
```

There is no structural difference between standard elements and elements intended as prototypes. Distinguishing prototypes from objects in this way is rather a mere “declaration of intent” aimed at helping the human reader understand the model.

These prototype elements serve as prototypes for *abstract* elements only. Prototypes for user interface elements must always be other user interface elements, as their visual appearance would otherwise be undefined. Prototypes for user interface elements are usually placed into a user interface design for the sole purpose of serving as prototypes at run time, and are usually made invisible to the end user.

III.3.6 Implicit Event Throwing

So far, the events we have introduced come directly from the user, such as mouse movement events. In addition, the EET interpreter will automatically generate three types of events for you as it interprets a model, namely events that notify you of the creation, deletion, or modification of an element. You can then attach transitions to these “implicit” events just like you would attach transitions to any other event.

For example, the EET interpreter generates an event when the length of a list-valued attribute is changed. It will then put the current event on a stack of events to be executed later, and first execute an artificial event like the following.

```
bb.changed(
          <string>  attr        "selectedPins",
          <integer> newLength   3)
```

This specific example tells us that the list-valued *selectedPins* attribute of the *bb* element just changed, and that it now contains three values. One can specify transitions on implicit events just as you would specify transitions on any other event.

For the sake of this example, imagine that the interface which we are modelling contains a Connect button. Its semantics are that it can be invoked only when there are exactly two pins selected. Thus, we want to enable the Connect button in this situation and keep it disabled otherwise. We can express this behavior using implicit events as follows.

```
Transition bb.changed(
     string attr == "selectedPins", int newLength != 2)
{
     ConnectButton.enabled := "false";
}

Transition bb.changed(
     string attr == "selectedPins", int newLength == 2)
{
     ConnectButton.enabled := "true";
}
```

Implicit events do not increase the computational power of EET models. For example, we could have also implemented the above behavior by inserting enabling and disabling statements into all transitions that select and de-select pins. However, the above implementation is preferable for many reasons.

First, it makes the situations which enable and disable the Connect button declarative and explicit, rather than hiding them within other transitions. This more declarative form is easier to understand for human readers. It is also more suitable for automated analysis components such as reachability checkers.

Second, if we later introduce new ways of selecting and de-selecting pins we will not have to add statements to the new transitions that take care of the enabling and disabling of the Connect button when pins are selected and de-selected in this alternative way.

Four standard implicit events are built into the EET interpreter. Table 3-1 lists the built-in implicit events.

Table 3-1: Standard Implicit Events

Situation	Example of Generated Event
An element is created.	`pin17.created()`
An element is deleted.	`pin17.deleted()`

A single-valued attribute is changed.

```
button3.changed(
    string attr == "enabled",
    boolean old == true,
    boolean new == false)
```

A multi-valued attribute is changed.

```
blackboard.changed(
    string attr == "selectedPins",
    integer oldLength == 2,
    integer newLength == 3)
```

All implicit events are generated and executed immediately after their triggering conditions occur. For example, the notification “pin17 has been created” is generated immediately after the corresponding create statement.

The sole exception is the “deleted” notification which is generated just *before* the element is actually deleted. This is because it is often convenient to refer to the attributes of an element that is about to be deleted in the intercepting transition - this would not be possible if it were already deleted at the time the notification is generated.

III.3.7 Explicit Event Throwing

The implicit events introduced in the previous section take care of most of the situations where one wants to execute a transition in response to a specific change to the elements. However, the Elements, Events & Transitions model also offers the facility to explicitly synthesize events (that is, the facility to manufacture artificial events from within an EET model). While this is rarely necessary within a single EET model, we will later see that this facility can also be used to communicate between several concurrently running EET models, and to communicate between an EET model and application code.

An event is explicitly synthesized, or “thrown”, by an event throw statement. Such a statement minimally contains the name of the event and the name of the object on which the event should occur. It can also additionally specify parameters.The transition below generates a *pinDeleted* event on the *blackboard* element.

```
Transition (*.isAPin=="true").deleted()
{
     throw blackboard.pinDeleted(
               string pinName := self.name);
}
```

There is no need to explicitly declare the events that may occur in an EET model up-front. We felt that while a separate explicit declaration may help prevent simple errors (such as misspelling an event name), they were not worth the additional complexity they would add to the language.

For reasons of uniformity, all events are required to state the element on which they occur. This is always meaningful for user interface events because they naturally occur on some user interface element.[21] Explicitly thrown events may not always be directly related to a particular element. In such cases, it is customary to specify an abstract element (see Section III.3.5) as the element that receives the event. For example, more complex EET models typically contain an abstract element that serves as a placeholder for various global variables, so that it can be used as the "receiving element" for an event unrelated to any particular element.

III.3.8 Integrating Application Code

In all of our previous discussion, we have dealt with a single Elements, Events & Transitions model. Thus, all events coming in from the user interface were sent to this model, and all implicitly and explicitly thrown events were processed by it.[22]

---

[21] This element is possibly the root window. Keyboard events are sent to the user interface element that is currently under the mouse cursor.

It is also possible to have multiple EET models and multiple applications running concurrently that communicate by sending each other events. In order to participate in this event communication, a component registers its unique name with the EET interpreter.[23]

It is then possible to send events to this component within an EET model. This is done by prefixing the element of the throw statement with the name of the receiving component. For example, assume that there is one EET model and one application running concurrently, and that the registered name of that application is “app”.[24] We can then send events to the application in the following fashion.

```
throw app::pin8.deleted();
```

Omitting the component specifier (“app” in the preceding example) - as was done in Section III.3.7 - indicates that the event will be processed by the

---

22. Implicit events are actually *always* exclusively processed by the local EET model for performance reasons. (If desired, you can explicitly forward these events to another model via event throw statements.)

23. The assumption here is that all models and applications run in a single address space, so that there is only one EET interpreter. If the EET models are distributed (running as separate processes not sharing memory) component names must be broadcast to all EET interpreters that may send events to these components.

model that threw the event. (The double-colon syntax is loosely related to its use in the C++ programming language, where it is also used for scoping.)

The element name (“pin8” above), event name (“deleted” above) as well as the types, names, and values of possible parameters are passed to the application as character strings to avoid platform dependencies in e.g the encoding of types and values (at the slight expense of efficiency).

There are many reasons for integrating custom application code with an EET model. The first reason, of course, is that it must be possible to invoke application functionality through the user interface. For example, if the user creates a new “folder” in a graphical application for file management, a piece of application code has to physically create a new file on disk by calling on operating system services.

Another reason to integrate custom application code is to implement complex, application-specific changes to the user interface. For example, assume that the user has designed a new circuit in a circuit design application and now presses a button which simulates execution of the circuit. In this case, it is often preferable for the application to directly update user interface elements for perfor-

---

24. We should mention here that registering and un-registering EET components can only be done via the Application Programmers Interface, and is not an activity intended for user interface designers.

mance reasons (rather than sending an EET event for every update of a user interface element that is then performed by the EET interpreter).

The final reason is to provide custom computation that is not built into the EET interpreter. The EET interpreter provides only the most common operators, such as concatenation of text strings, and addition and multiplication of numbers.[25] More complex functions - such as a co-sine function, a logarithm function, or an application-specific function - can be implemented in application code. The EET model then sends an event requesting this computation to the application, including any argument values that may be required. The application returns the resulting values by either sending them as parameters of another event or by placing them into designated attributes where they can be accessed from the EET model.

Section III.3.9 will present an EET-based application in great detail, and Appendix C will provide an example of actual C++ code that is integrated with an EET model.

III.3.9 Example: A "Nodes and Links" Application

We conclude the section on the advanced concepts of the Elements, Events & Transitions model with another example of an EET-driven user interface.

---

[25] Providing a large number of rarely used functions would unnecessarily bloat the size of the interpreter, and it is obviously impossible to anticipate application-specific functions.

This example is significantly more complex than the introductory example in Section III.2.4. For example, it involves objects that are instantiated and deleted at run-time, and it communicates with application code. Its EET model intentionally makes use of many of the advanced concepts discussed in Section III.3, among them statements that operate on sets of elements (Section III.3.1), transitions on a class of elements (Section III.3.2), transitions that create and delete elements (Section III.3.3), multiple indirection in attribute references (Section III.3.4), and abstract elements (Section III.3.5). It also uses implicit event throwing extensively (Section III.3.6), and throws explicit events to communicate with application code (Sections III.3.7 and III.3.8).

Figure 3-2 shows the user interface of our example, a simple graph editor that we will call the “Nodes and Links” application.

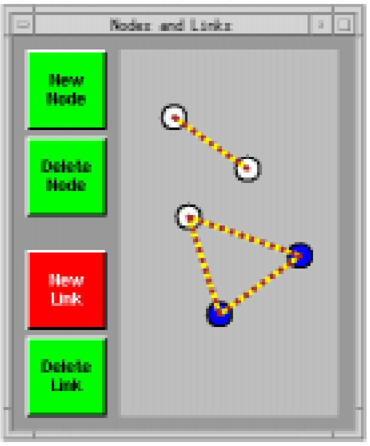


Figure 3-2: The “Nodes and Links” Application in Use

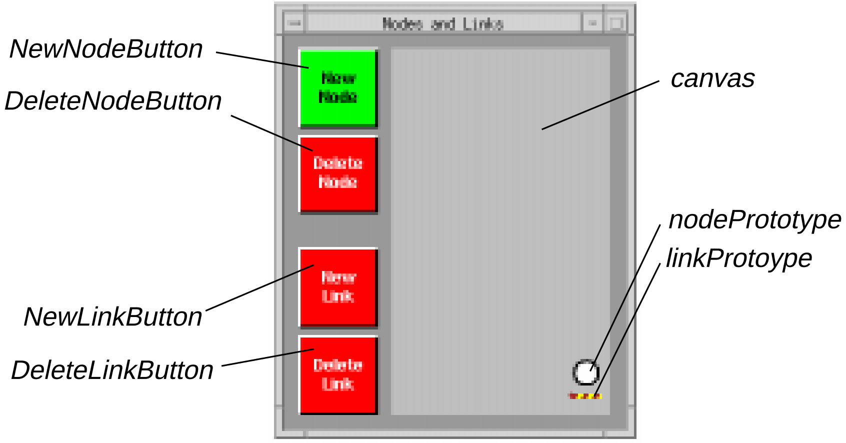
The Nodes and Links application lets the user create and delete nodes as well as links between those nodes. The links stay attached to the nodes when the nodes are moved. The selection of any node can be toggled by clicking on it. Clicking on the canvas de-selects all nodes. The command buttons are enabled and disabled as needed to show their applicability. The New Node button is always enabled. The Delete Node button is enabled if there is at least one currently selected node. The New Link button is enabled if exactly two nodes are selected and there is not a link between them already. Finally, the Delete Link button is enabled if exactly two nodes are selected and there *is* a link between these two nodes.

We will now describe the EET model that drives this interface in great detail. A user[26] will normally start a new design by laying out the initial interface, as shown in Figure 3-3.

In this section, we will exclusively focus on the modelling constructs of the EET model, and not present the evolution of this design. Thus, we will list the complete EET model of the Nodes and Links application, interspersed with our

---

[26] In our terminology, any user of our environment is a “user”. Thus, the “user” is typically an interface designer or a programmer. We use the term “end-user” to describe the intended audience of EET-driven applications (which normally neither knows nor cares if the application is EET-driven or implemented conventionally).



Note that the two prototype objects in the lower right corner are normally invisible to the end user. They are visible here only to demonstrate how this application is implemented.

Figure 3-3: Initial Layout of the “Nodes and Links” Application

comments. The model consists of one abstract element and seventeen transitions.

```
Object bb {
     <int>          xOfNextNewNode 10;
     <int>          yOfNextNewNode 10;
     <string>       currentlySelectedNodes ;
}
```

Section III.3.5 introduced the notion of elements that have no immediate representation in the user interface. The above object is such an element. Its name is a shorthand for a “blackboard” element which we use to store data that does not belong to any user interface element in particular. The two integer attributes *xOfNextNewNode* and *yOfNextNewNode* will be used to store the position on the canvas where a new node will appear. The *currentlySelectedNodes* attribute will hold the list of the names of the currently selected nodes.

```
#1
Transition (*.isAButton=="true").enable()
```

```
{
     self.status := "enabled";
     self.textForeground := "black";
     self.fillForeground := "green";
}

#2
Transition (*.isAButton=="true").disable()
{
     self.status := "disabled";
     self.textForeground := "white";
     self.fillForeground := "red";
}

#3
Transition ((*.isAButton=="true") &&
            (*.status=="enabled")).released()
{
     throw self.invoke();
}
```

Transitions #1-3 implement an enabling/disabling functionality for command buttons. Implementing user interface behavior on this level is only necessary if the underlying toolkit does not already provide this functionality, or when a custom look and feel for buttons is desired.

In this implementation, transitions #1 and #2 specify that enabled buttons are green with black text while disabled buttons are red with white text. We have attached a custom attribute called *status* to the buttons, whose value is either *enabled* or *disabled*. This allows transition #3 and others to refer to “disabled” buttons rather than “red” buttons, making the rest of the design independent of the chosen colors. We have also attached an attribute called *isAButton* to all buttons which allows us to conveniently differentiate between button elements and others. We will later use the same technique to mark node elements and link elements.

Transition #3 uses explicit event throwing (Section III.3.7) to generate an *invoke* event on a button when it is clicked while being enabled.

```
#4
Transition ((*.isANode=="true") &&
            (*.status=="not selected")).pressed()
{
     throw self.select();
}

#5
Transition ((*.isANode=="true") &&
            (*.status=="selected")).pressed()
{
     throw self.deselect();
}

#6
Transition (*.isANode=="true").select()
{
     self.status := "selected";                          #a
     self.fillForeground := "blue";                      #b
     bb.currentlySelectedNodes += self.name;             #c
     throw app::nodesAndLinks.checkLinkButtons();      #d
}

#7
Transition (*.isANode=="true").deselect()
{
     self.status := "not selected";
     self.fillForeground := "white";
     bb.currentlySelectedNodes -= self.name;
     throw app::nodesAndLinks.checkLinkButtons();
}
```

Transitions #4 through #7 describe how nodes are selected and de-selected. Transitions #4 and #5 express that a *select* event is thrown if the user clicks on an element which is not selected, and that a *deselect* event is thrown if it is already selected - the selection status of nodes toggles at every click. Transitions #6 and #7 then specify what happens upon these *select* and *deselect* events. A custom attribute called *status* marks the node as “selected” or as “not selected” (statement #a). The foreground color of a node changes from white to blue when it is selected (statement #b). We also keep track of the selection status of the nodes in the *currentlySelectedNodes* attribute that belongs to the abstract “bb” blackboard element we discussed earlier (statement #c). Finally, statement #d uses explicit event throwing to invoke application code that will be discussed later in this section.

```
#8
Transition (*.isANode=="true").motion(int x, int y)
{
	self.x := x;
	self.y := y;
	throw ((*.from==self.name) ||
(*.to==self.name)).adjust();
}
```

Transition #8 describes how users can move nodes, namely by dragging them. This transition - as well as any other transition dealing directly with events from the user interface - is highly dependent on the kind of events that are sent from the user interface. In the particular toolkit we have used for our proof-of-concept implementation, pressing a mouse button over an element and then moving the mouse results in *motion* events belonging to this element. These events contain the mouse position as parameters (as well as other information such as the status of modifier keys). Thus, the two assignment statements of transition #8 implement dragging for nodes. The third statement then causes *adjust* events on links attached to this node, so that the links always stay attached to their nodes, even while a node is being dragged. The throw statement can be paraphrased in English as “adjust all links that refer to myself in a *from* attribute or in a *to* attribute”.

```
#9
Transition canvas.pressed()
{
       throw (*.status=="selected").deselect();
}
```

Transition #9 specifies that all currently selected elements are de-selected if the user clicks on the canvas (the *canvas* object is the layout parent of all nodes and links as illustrated in Figure 3-3).

```
#10
Transition (*.isANode=="true").deleted()
{
       bb.currentlySelectedNodes -= self.name;
       delete ((*.from==self.name) ||
               (*.to==self.name));
}
```

Transition #10 makes use of the implicit deleted event that is thrown just before any element in am EET model is deleted (Section III.3.6). In this case, the first statement of transition #10 removes the node’s name from the list of selected elements. The second statement then deletes all links that are attached to the node.

```
#11
Transition (*.isALink==1).adjust()
{
  self.x0 := (self.from.x + ((1 / 2) * self.from.width));
  self.y0 := (self.from.y + ((1 / 2) * self.from.height));
  self.x1 := (self.to.x + ((1 / 2) * self.to.width));
  self.y1 := (self.to.y + ((1 / 2) * self.to.height));
}
```

Transition #11 specifies what happens in response to *adjust* events on links that are thrown by transition #8 and others. The four assignment statements compute the position of the end-points of a link so that they are centered on the nodes. The assignments make use of multiple indirection in attribute referencing (Section III.3.4) to access the position and dimensions of the two nodes that they are attached to.

The next four transitions specify the behavior of the buttons.

```
#12
Transition NewNodeButton.invoke()
{
     element n := create nodePrototype;               #a
     n.x := bb.xOfNextNewNode;                          #b
     n.y := bb.yOfNextNewNode;                          #c
     bb.xOfNextNewNode := (bb.xOfNextNewNode + 20);     #d
     bb.yOfNextNewNode := (bb.yOfNextNewNode + 30);     #e
     n.mapped := 1;                                     #f
     throw n.select();                                  #g
     throw app::nodesAndLinks.checkLinkButtons();       #h
}
```

The transition for the “New Node” button first creates a new node by copying from the prototype for nodes (statement #a, the node prototype is shown in Figure 3-3). Like all create statements, this statement provides a symbolic identifier, named *n* here, which can be used to refer to the new element in the remaining statements of this transition; symbolic identifiers are necessary here because we do not know the (automatically generated) name of a new element at this point.

The next four statements then set the position of the new node, and specify that the next new node will be created at an offset of this one (statements #b-#e). Note that this implementation of creating new elements at an offset from each other is simplistic here for brevity. The position of new nodes will be further and further to the right and bottom of the canvas area - until they are finally located outside the visible canvas area if many nodes are created. The problem can be addressed by re-using old positions that are no longer used.

Statement #f then makes the new element visible - the underlying toolkit uses a numeric *mapped* attribute to control visibility. Explicitly making the element visible is necessary because the node prototype is normally invisible.

Finally, statement #g implements our design decision that new elements are automatically selected, and statement #h informs application code that a new element has been created. The following three transitions also throw a *checkLinkButtons* event to the application.

```
#13
Transition DeleteNodeButton.invoke()
{
       delete ((*.isANode=="true") &&
               (*.status=="selected"));
       throw app::nodesAndLinks.checkLinkButtons();
}
```

Transition #13 specifies that pressing the Delete Node button will delete all currently selected nodes - deleting nodes will then automatically trigger further action via the implicit delete events (transition #10).

```
#14
Transition NewLinkButton.invoke()
{
     element n := create linkPrototype;
     n.from := bb.currentlySelectedNodes[1].name;
     n.to := bb.currentlySelectedNodes[2].name;
     throw n.adjust();
     n.mapped := 1;
     throw app::nodesAndLinks.checkLinkButtons();
}
```

Pressing the New Link button is possible only when exactly two nodes are selected, and when these nodes are not already connected by a link (the logic for enabling and disabling the New Link and Delete Link buttons is implemented in application code as described later). Transition #14 first creates a new link element and sets its custom *from* and *to* attributes to the selected nodes. (The *from* element is the one that the user selected first, but this is not really relevant in this application as all links are bidirectional.) The transition then uses explicit event throwing to set the end points of the link to the centers of its nodes, and makes the link visible.

```
#15
Transition DeleteLinkButton.invoke()
{
  delete (((*.from==bb.currentlySelectedNodes[1].name) &&
           (*.to==bb.currentlySelectedNodes[2].name)) ||
            ((*.from==bb.currentlySelectedNodes[2].name)
&&
                (*.to==bb.currentlySelectedNodes[1].name)));
    throw app::nodesAndLinks.checkLinkButtons();
  }
```

Similar to the New Link button, the Delete Link button can only be invoked when exactly two nodes are selected that are connected by a link. Transition #15 deletes that link in response to a click on the Delete Link button. Its delete statement can be translated as “delete the link whose *to* attribute matches the first selected node and whose *from* attribute matches the second selected node, or vice versa”.

```
#16
Transition bb.changed(
     string attr == "currentlySelectedNodes",
     int newLength > 0)
{
     throw DeleteNodeButton.enable();
}

#17
Transition bb.changed(
     string attr == "currentlySelectedNodes",
     int newLength == 0)
{
     throw DeleteNodeButton.disable();
}
```

Finally, transitions #16 and #17 describe under which conditions the Delete Node button is enabled, namely when at least one node is selected. Their use of the implicitly thrown “changed” event (Section III.3.6) makes them independent from the transitions that actually change this attribute. This way, these two transitions do not have to be touched when e.g. new methods for selecting nodes are added.

The last missing piece of functionality is the description of when the New Link and Delete Link buttons become enabled and when they become disabled. This functionality is implemented in application code. Transitions #6-#7 and #12-#15 throw *checkLinkButtons* events to an EET component called *app*. This specifier denotes application code - a programming language procedure is called that contains the event as well as pointers to the state of the EET model. The programming language can then be used to do arbitrary computation, and to affect the state of the EET model by either changing elements directly, or by sending events back to the EET model.

In our case, the application responds to *checkLinkButtons* events by sending *enable* or *disable* events for the New Link and Delete Link buttons as necessary. It accesses the current state of the elements in order to decide whether or not to change the enabling of the buttons, but it does not change the state of any element.

In our case, the functionality of the application code is simple: send an *enable* event for the New Link button if the button is currently disabled and exactly two nodes are selected which are not connected yet, send an *enable* event for the Delete Link button of this button is disabled and exactly two nodes are selected which are connected (and do the reverse to disable them as necessary).

The conditions that a button is currently disabled and that there must be exactly two nodes selected could easily be captured within an EET expression (assuming that we maintain a list that keeps track of the currently selected elements, as in the example in Section III.3.6, or that we alternatively at least maintain a variable that keeps track of the number of selected elements). The part that cannot be captured within a single EET expression is that the two selected nodes are not connected by a link. This is because EET expressions only provide the lowest-order logic operators (*and*, *or*, and *not*) but not the higher-order quantifiers *there exists* and *for all*. A language extension is possible, and would make some application code unnecessary. In this particular case, we could then write a condition of the form “the button is disabled, and there are two selected nodes *n1* and *n2*, and there does not exist a link l such that *l.from*==*n1* and *l.to*==*n2* or such that *l.from*==*n2* and *l.from*==*n1*”. We felt that there were too few cases where quantifiers make application code unnecessary to warrant extending the expressions language, and that writing expressions of that complexity would be as difficult as writing equivalent application code.

Appendix C lists the actual code that implements this functionality.

III.4 Using Two or More Levels of Abstraction

Section III.2 introduced the building blocks of an Elements, Events & Transitions model, and Section III.3 discussed some advanced features for structuring models. However, the preceding two sections have only dealt with models at a single level of abstraction This section concerns using two or more EET models to describe functionality at multiple levels of abstraction.

The Elements, Events & Transitions model supports the use of arbitrarily many levels of abstraction. In practice, there is rarely a need to go beyond two levels of abstraction even for large designs, and almost never a need to go beyond three. Increasing the number of abstraction levels in an EET-based application results in a more modular design while at the same time requiring a higher intellectual effort from the designer or programmer. For the latter reason, it is often preferable to start out with a single level of abstraction, and to introduce a second level only later in the design.

III.4.1 Why Use Models at Multiple Levels of Abstraction?

One motivation is to make a design less dependent on the concrete events coming from the user interface. This will help in porting designs to a new platform where the events sent from the user interface will typically differ. Used this way, a lower-level EET model does little more than translate low-level incoming events such as “delete button pressed” into higher-level events such as “delete command invoked”. The latter event is than processed by a separate higher-level model.[27]

A similar motivation is to use the same higher-level model to explore several user interface alternatives on the same platform. For example, the higher-

---

[27] In a previous system, “user interface tasks” served as a lower-level user interface model while “application tasks” were used at a higher level [Suka93]. There is a rough but not perfect correspondence to using two EET models at different levels of abstraction.

level model of a file management application may have a command to delete a group of files. A designer can then hook up several alternative user interface designs to this command by sending a corresponding EET event to the higher-level model.

Finally, separating aspects specific to a user interface platform from aspects specific to a user interface paradigm will help automated components analyze the model. For example, a consistency and completeness checker can then operate on the higher level without having to deal with platform-specific aspects of the model.

III.4.2 Organization Alternatives

This section provides block diagrams of the most typical organization alternatives for EET-based applications. The simplest possible organization is a single EET model with no attached application code, as illustrated in Figure 3-4.

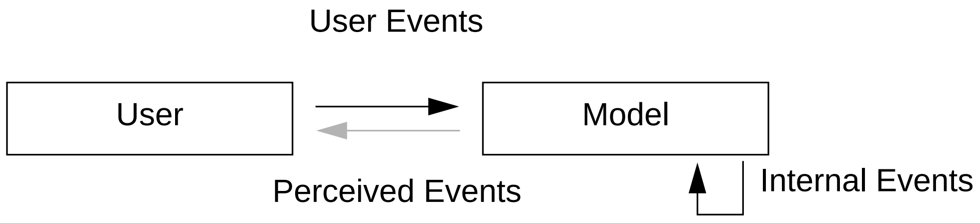


Figure 3-4: The Simplest Use of an EET Model

In this organization, the user causes events that are forwarded from the platform-specific user interface front-end to the EET interpreter. The EET interpreter than processes the transitions triggered by this event, possibly causing several more internal events to be generated and processed. We will omit internal events in the subsequent diagrams.

For completeness, we will mention here that changes to the user interface elements caused by the EET interpreter will often be perceived as an “event” by the designer and end-user, even though no EET events were sent. For example, the EET interpreter may raise a mailbox flag by manipulating element attributes. This action will be perceived as an event by the end-user (“I got mail!”) but there is no EET event to the user interface front-end involved. Instead, the change in appearance is achieved by manipulating the element space.

The above organization can be used for user interface prototypes that do not actually invoke application functionality. The simplest way to integrate application code with an EET model is shown in Figure 3-5.

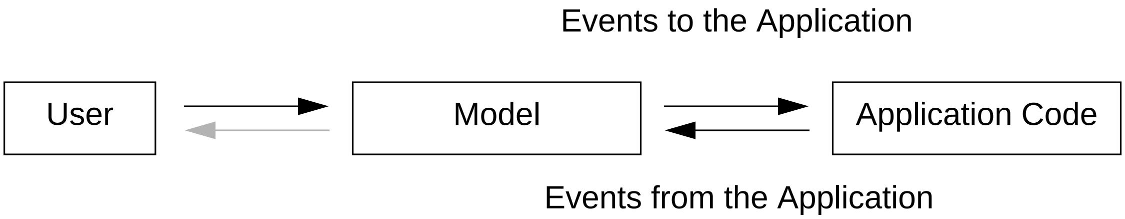


Figure 3-5: A Single EET Model and Application Code

In this organization alternative, the interaction between the user and the EET model is the same as above, but the EET model additionally invokes application code by throwing explicit events as described in Sections III.3.7 and III.3.8.

The application can in turn make changes to the element space by either sending events back to the EET model or by directly changing elements.

Finally, Figure 3-6 shows the use of two EET models at different levels of abstraction. In this organization, events from the user are first sent to the lower-level EET model, which translates them into semantically more meaningful events sent to the higher-level EET model. For example, a mouse-down event on an object may be translated into a “selection” event. The higher-level will in turn send events to the lower-level model such as “de-select all objects”. This scheme is similar to one used in the second-generation UIDE [Suka93].

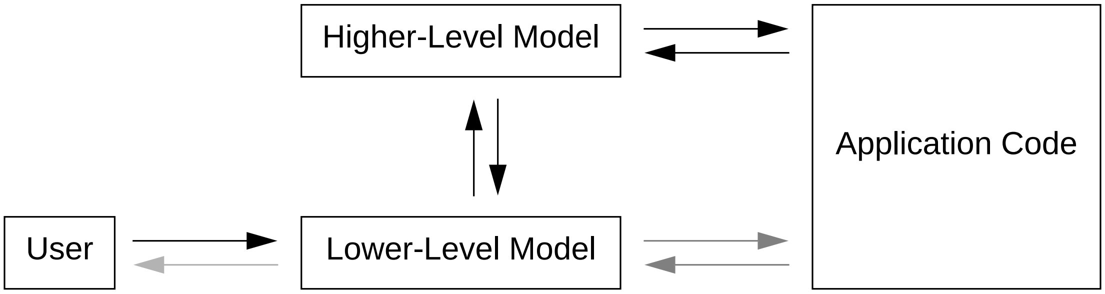


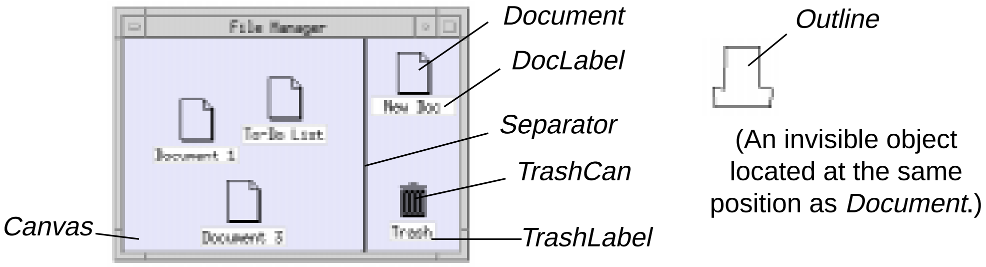
Figure 3-6: Two EET Models at Different Levels of Abstraction

The higher-level EET model will then communicate with the application as discussed above. This form of organization ensures that both the higher-level EET model and its attached application code can be moved to a different platform by replacing the lower-level, platform-specific model.

The lower-level model can also directly communicate with the application code. This is undesirable because it tightly couples this application code to platform-specific events and elements, which is why the corresponding arrows are dashed in Figure 3-6. However, it is often unavoidable to tie some application code to low-level events in highly interactive applications (e.g. for semantic feedback during low-level interactions). In general, as little application code as possible should be tied directly to a low-level EET model, but as much as needed.

III.4.3 Example: A File Management Application

Figure 3-7 shows a file management application that we will use as an example of an EET-based application using two levels of abstraction.



(An invisible object located at the same position as *Document*.)

Figure 3-7: A File Management Application

It provides for the creation of new documents by dragging and dropping a palette document (labelled “New Doc” in Figure 3-7), for their deletion by dragging to a “trash can”, and for their renaming by editing labels. The system assigns automatically generated names to new documents. In this particular case, the end-user has created three new documents, and has then renamed document “Document 2” to “To-Do List”.

We will first present the lower-level EET model that implements this behavior (without describing it in as much detail as earlier in this chapter). We will then explain how it communicates with a higher-level EET model.

```
Transition Document.pressed()                    LOW #1
{
        element n := create Outline;
        n.mapped := 1;
        n.current := "yes";
}

Transition Document.moved(                       LOW #2
        coord x, coord y)
{
        (*.current=="yes").x := x;
        (*.current=="yes").y := y;
}

Transition Document.released()                   LOW #3
{
        delete (*.current=="yes");
}
```

The first transition makes an outline of a new document appear when the user presses on the “New Doc” icon in the palette. The second and third transitions then make it follow the mouse until the mouse button is released, at which point the outline document disappears. The attribute “current” is a user-defined string attribute that makes it convenient to refer to the current outline object.

```
Transition Document.released(                    LOW #4
        string o == "Canvas",
        coord x < Separator.x,
        coord y)
{
        element n := create Document;
        n.x := x;
        n.y := y;
        element n2 := create DocLabel;
        n2.doc := n.name;
        n2.textMode := "editInsert";
        throw (*.name==n2.name).adjust();          #a
        throw app::fileManager.provideLabelFor(    #b
                string label := n2.name);
        throw amodel::fileManager.newDocument(     #c
                string id := n.name,
                string name := self.text);
}
```

If the outline document is released over the main area of the canvas (left of the separating line, that is), a new labelled document appears there. The “doc” attribute of the document label is another user-defined string attribute - it is used to keep track of which document a label belongs to. The “textMode” attribute for text fields is built into the toolkit used by our prototype implementation, and enables editing its text when set to “editInsert” (rather than “editReadOnly”).

The first throw statement (#a) is used to invoke an auxiliary transition that provides the computation for properly placing a document label under its icon, and is used similar to a sub-routine call in a procedural programming language. The second throw statement (#b) invokes an application routine for computing the name of a new document (“Document 1”, “Document 2”, and so on). The third throw statement (#c) notifies the higher-level EET model that a new document was created.

```
Transition Canvas.setLabelFromApp(                LOW #5
        string e, string t)
{
        (*.name==e).text := t;
}
```

The above transition (#5) is called exclusively from the application, and is used to set the text of a document label (it is invoked from the “provideLabelFor” application routine discussed above).

```
Transition (*.class=="TextField").adjust()        LOW #6
{
    self.x := (self.doc.x -
               ((1 / 2) * (self.width - self.doc.width)));
    self.y := ((self.doc.y + self.doc.width) + 2);
}
```

This transition (#6) provides the computation for properly placing a label under the document as mentioned earlier.

```
Transition (*.class=="TextField").keypressed(     LOW #7
        string k == "Return")
{
        throw amodel::fileManager.renameDocument(
                string id := self.doc.name,
                string newName := self.text);
}
```

The toolkit used in our prototype implementation takes care of low-level editing operations on text fields, such as inserting characters as they are typed, moving the cursor back and forth with the arrow keys, deleting characters with the Delete key, and so on.

However, each individual key press event is then nevertheless also processed by the EET interpreter so that one can attach additional behavior to keyboard events. We use this mechanism in transition #7 to notify the higher-level EET model when the editing operation is complete (which is when the “Return” key is pressed).

```
Transition ((*.class=="DocIcon") &&                LOW #8
            (*.name!="Document")).moved(
        coord x, coord y)
{
        self.x := x;
        self.y := y;
        throw (*.doc==self.name).adjust();
}
```

Transition #8 makes all documents draggable with the mouse (with the exception of the palette document itself). The “adjust” transition (transition #6) is re-used here for making the label move with its document icon.

```
Transition (*.class=="DocIcon").released(          LOW #9
        string o == "TrashCan")
{
        throw amodel::fileManager.deleteDocument(
                string id := self.name);
        delete (*.doc==self.name);
        delete self;
}
```

Finally, transition #9 deletes a document icon that is released over the trash can (and its label), but not before notifying the higher-level EET model.

This ends our discussion of the lower-level EET model, which implements all the details of the user interface. It notifies the higher-level model in three places, namely when documents are created, deleted, or renamed (in transitions #4, #7 and #9). The higher-level model is shown below.

```
Object fileManager {
}

Transition fileManager.newDocument(               HIGH #1
        string id, string name)
{
        throw app::fileManager.newDocument(
                string i := id,
                string n := name);
}

Transition fileManager.renameDocument(          HIGH #2
        string id, string newName)
{
        throw app::fileManager.renameDocument(
                string i := id,
                string n := newName);
}

Transition fileManager.deleteDocument(          HIGH #3
                string id)
{
        throw app::fileManager.deleteDocument(
                string i := id);
}
```

EET events always occur on an element, while the three notification events above are not related to any element in particular. We have therefore introduced the abstract element “fileManager” for the sole purpose of serving as the element on which the events occur (a technique discussed in Section III.3.7).

In this example, the higher-level EET model does little more than passing events through to the application. It is nevertheless useful in providing a definition of the user-level operations that the lower-level EET model has to provide, and in separating high-level application routines from application routines closely tied to a particular user interface style (as discussed in Section III.4.1). In this case, the “provideLabelFor” routine tied to transition #4 of the lower-level model is specific to the user interface style used, but the three functions called from the high-level model are not.

III.5 Discussion of the EET Model

We conclude the chapter with a discussion of the EET model's role as the basis for demonstrational tools and of its role in user interface design in general.

III.5.1 Parallel Execution

The primary motivation for the EET language is its support for demonstrational tools. Our demonstrational tools use "before" and "after" snapshots to describe behavior (as we will explain in detail in Chapter IV). Conceptually, the changes that transform a "before" snapshot to its corresponding "after" snapshot occur at the same time.

For example, imagine that selecting a "Properties" menu item both (a) dims this menu item and (b) brings up a new window. The end-user will perceive these two changes as happening in parallel (be the actual computing platform a single-processor or a multi-processor machine) because they are executed so quickly that they cannot be told apart.

The EET interpreter executes blocks of assignment statements "conceptually in parallel". That is, it first evaluates the right-hand sides of all assignments before it actually changes any elements.[28]

---

[28] The interpreter also pre-evaluates the left-hand side expressions of set-valued assignment statements such as "(*.mapped==1).color := ..." because other assignment statements may affect which elements match its expression.

Imagine that there are two elements *a* and *b* which switch color upon a certain event. With conceptual parallelism, we can express this behavior as follows.

```
a.color := b.color;
b.color := a.color;
```

We could otherwise not express this functionality in two lines (we would need an auxiliary variable and another assignment). Even more importantly, this language fragment more closely corresponds to the actual demonstration (which has no notion of auxiliary variables, see Chapter IV). Finally, conceptual parallelism in the underlying language allows our demonstrational reasoning engine to treat each change from a “before” to an “after” snapshot independently of others (which is computationally desirable).

We should stress here that only blocks of consecutive assignment statements are executed in parallel. Consider the following EET language fragment.

```
element n := create obj1;
n.attribute1 := 1;
n.attribute2 := 2;
throw obj2.event1(string par1 := n.attribute1);
n.attribute3 := 3;
```

The create statement is executed first, then the two subsequent assignment statements (in parallel), then the throw statement, and then the final assignment statement.

The reasons for not providing conceptual parallelism for other statements are as follows. It is obvious that create statements must be executed before subsequent assignments referring to the newly created object. It is also often convenient to be able to refer to the results of earlier computation within an event throw statement, and the same is true for delete statements to a lesser extent.

There are good reasons both for and against parallelism in executing transitions. We believe that the current compromise of only executing assignment statements in parallel provides for a good balance between supporting the demonstrational tools (statements are parallel so that they can be considered independently) and supporting the manual generation of the language (statements are sequential so that each statement can refer to earlier computation within the same transition).

III.5.2 Performance Issues

The purpose of the EET model is to facilitate the interactive design of user interfaces. We therefore implemented an interpreter for the EET language so that the designer can instantaneously switch from design mode to test-drive mode and vice versa, without having to compile and link code.

Any language that is interpreted at run-time incurs some overhead compared to a language that is pre-compiled into a micro-processor’s native code. In addition, some features of the EET language, such as implicit event throwing, are potentially expensive for large EET models.

There are three fundamental ways of improving the performance of EET-based applications.

III.5.2.1 Improving Performance by Smarter Interpretation

The proof-of-concept prototype of the interpreter does not make use of smart optimization techniques. For example, an improved version could construct indices that map an event type to the transitions it can potentially trigger. (These indices can be built when the EET model is read in; they do not change at run-time.) The EET interpreter then does not have to evaluate all transition invocation conditions for every event. It can rather just evaluate the transitions that may be affected by consulting the index. Thus, maintaining such indices should dramatically speed up performance for large EET models.

III.5.2.2 Improving Performance through Compilation

An obvious way of improving performance is by providing an EET compiler in addition to the interpreter. The designer can then use the interpreter for interactive design, and use the compiler to produce the finished version to be delivered to customers.

III.5.2.3 Improving Performance by Model Restructuring

Performance can also be improved by restructuring models, and by moving more of the event processing to application code. In contrast to improvements to the EET interpreter, restructuring models puts a burden on the EET users. This is desirable only in the later phases of the design.

A typical candidate for optimization is the processing of events that occur with high frequency, such as mouse movement events. The responsiveness of dragging can be improved by moving the processing of these events into custom application code. These optimizations should only be performed at the latest stages of a design, when actual application code has already been attached, and when it is unlikely that the user interface will change significantly in the future.

III.5.3 EET Models and Object-Orientation

The Elements, Events & Transitions model strictly separates the description of static properties of a user interface (contained in *elements*) from the description of its dynamic properties (contained in *transitions*). This differs from an object-oriented model where objects contain the description of both their static properties and their dynamic properties.

The advantage of strictly separating those is that the designer can specify the static properties first (by direct manipulation) and the dynamic properties later (by interactive demonstration). Another advantage for our purposes is that all description of behavior is in one place, and that it is easy to specify transitions such as “when any button is pressed” without having to introduce the user to the concept of classes.

This separation does not imply that e.g. our demonstrational technology could not be used in an object-oriented environment. It would also be straightforward to make EET models themselves more “object-oriented.” Transitions that apply only to element *x* can naturally be grouped with element *x* itself, so that they become *object methods*. Transitions that apply to a class of objects can similarly be grouped with that class, so that these transitions become *class methods*. (The remaining transitions - those with complex invocation conditions - remain global methods detached from particular objects or classes.)

III.5.4 Modularizing Large EET Models

We chose to use a single EET model for all aspects of the user interface because it is convenient to have all behavior in a single place. However, it is also possible to break up the behavior description into several EET models. Doing so should improve performance because the interpreter then has to match fewer transitions against incoming events. This is especially desirable for large models.

Breaking up a large model into several smaller ones in this way does not require any change to the language as it is described in this chapter. This can rather be accomplished by simply registering additional participants for the event communication.[29] Three participants are pre-registered in our prototype implementation, namely the interface model (“imodel”), the application model (“amodel”), and optional custom application code (“app”). Figure 3-6 shows the organization diagram for this standard organization. The mechanism for sending events between participants is described in Section III.3.7.

---

[29] Registering and un-registering EET components is done from programming language code through the Application Programmer Interface (API) to the EET library, and is intended for programmers, not user interface designers.

There are several ways of using finer-grained EET models. The most radical solution is to register each element as an event client when it is created. In this case, all of its attributes are set by sending events rather than by accessing its state directly. The set of events defined on an element then serves as its external interface. However, breaking up the model into so many parts may actually slow down execution because even the simplest state changes require event communication.

The best solution may be to have one model per window. This way, the individual models are large enough for internal state changes to not require event communication, while not being so large that matching incoming events against its transitions becomes expensive.

III.5.5 Comparison of the EET Model and the Relational Model

There is a rough analogy between the Elements, Events & Transitions model for user interfaces and the relational model for databases [Codd70], and there is a similar analogy between the EET behavior description language and the SQL database query language [Astr75].

Both the relational database model and the EET user interface model impose more structure on their data than general-purpose programming languages. The relational database model imposes a tabular structure, and also imposes some restrictions on the content of the data (it has to comply with some “normal form” [e.g. Ullm83, section 7.4]). The EET model also imposes structure on user interface elements: it must be possible to describe them as a collection of attributes.

For both models, the implications are that (a) there are situations where a more general approach is preferable because the data cannot naturally be expressed in the respective model but that (b) the model otherwise allows high-level operations which take advantage of this special data structure.

In the case of the relational database model, there are situations when it is preferable to use an object-oriented database because the data to be stored cannot naturally be expressed in the relational model (e.g. engineering and scientific data, sound, pictures). However, in the many cases where the data can be mapped to tables, the relational model greatly simplifies the management of data.

In the case of the Elements, Events & Transitions model, there are also situations where it is preferable to use a more general model for structuring user interfaces, such as custom code in a general-purpose programming language (if the data to be presented does not map naturally to elements, such as large amounts of text, the contents of a video image, and so on[30]). However, in the many cases where user interface objects can be mapped to EET elements, we

---

[30] However, the EET model and language can of course still be used to drive the remaining aspects of the user interface, such as controls, windows, and canvasses containing icons.

can make use of the special structure to provide a simple interpreted language for the manipulation of these elements, as well as interactive tools.

Finally, it is worthwhile to note that neither language is computationally complete. For example, neither can compute a transitive closure (“starting from this element, find all elements that it refers to, and all elements that they refer to, and so on, until you have found all of them”). Both languages become computationally complete only by embedding their statements in a general-purpose programming language. For both languages, some have argued that the use of a special-purpose language is not justified because they are subsumed by general-purpose programming languages. We feel that the special-purpose languages are justified because they greatly simplify the management of the domains to which they are applicable.[31]

III.5.6 Comparison of the EET and Event-Response Languages

Ralph Hill’s Sassafras user interface management system [Hill86] includes a behavioral description language called the Event-Response Language (ERL). This language models the system’s reaction to user actions as responses to

---

[31] Sometimes, an alternative is to use a subset of an existing language. (However, SQL is not really a subset of any existing language to our knowledge; and we are also not aware of a language that would provide the equivalent of our “set expressions”, especially for the invocation of transitions.)

events, similar to our “transitions”, so that it is worth comparing and contrasting these languages.

The Event-Response Language’s primary design goal was to capture the human-computer dialog for multiple input streams (e.g. originating from two mice, for example).[32] The ERL addresses the dialog only, operations such as object creations and other changes to the user interface have to be implemented in programming language code. This was viable because Sassafras was intended for advanced programmers.[33]

The Elements, Events & Transitions language’s primary design goal is its use with interactive specification tools. Its focus group are computer-literate user interface designers who do not necessarily have implementation experience. This focus is reflected in the language by including simple interface-level operations

---

[32] It is worth noting that most modern windowing systems [e.g. Sche86] solve the multiple input stream problem by simply serializing the events into a single input queue. This solution appears superior because it facilitates multiple input devices for the user without exposing the system implementer to the difficulties inherent in parallelism (deadlock, race conditions).

[33] Citing from [Hill86], page 194: “Sassafras ... relies heavily on the tools provided by the Interlisp-D environment. Hence, many of the supporting tools need further development and currently are only usable by experienced Interlisp-D users.”

into the EET language, so that users can express simple user interface behavior without having to resort to a general-purpose programming language.

Another interesting difference between the languages are their invocation mechanisms for system actions (“rules” in the Event-Response Language, “transitions” in the EET language). ERL rules fire either in response to atomic events or spontaneously (spontaneous firings correspond to ε-rules in automata theory). Their firing can depend on the state of a list of boolean variables. The EET language provides for a more expressive invocation mechanism, namely for set expressions (Section III.3.2).[34]

This concludes our discussion of the Elements, Events & Transitions model.

---

[34] The obvious trade-off is that the interpreter then has to evaluate these expressions for each incoming event at run-time (see Section III.5.2).

CHAPTER IV

BUILDING MODELS BY DEMONSTRATION

The preceding chapter has discussed the Elements, Events & Transitions model, and has presented its textual syntax in detail. There are many good reasons for using a textual specification language to describe user interface behavior. For example, text can be efficiently stored, is well suited to describe complex behavior, consumes little screen space, and provides a readable, printable, editable, and shareable representation of dynamic behavior.

However, a design environment that relies *exclusively* on textual behavior description has a significant shortcoming. Namely, it requires considerable training to be able to translate desired user interface behavior into its textual description because there is no immediately obvious mapping between interactive behavior and text. Potential new users can easily be intimidated by an all-textual specification language, and may consequently reject the design environment as a whole.

We address this problem by providing demonstrational tools that automatically translate interactive behavior into the appropriate textual form. This enables novice users to specify behavior interactively, without having to learn any textual constructs beforehand. Even more importantly, users can learn the textual modelling language “on the fly” by inspecting the specifications generated from their demonstrations.

This chapter is subdivided into five sections. Section IV.1, IV.2 and IV.3 will present the user’s view of the demonstrational components, and will present many examples of their use. Section IV.4 will then describe the algorithms that the components use internally to interpret demonstrations. Section IV.5 concludes.

All programming-by-demonstration systems provide some particular method for the user to input demonstrations. This method must be formal enough for the system to be able to draw meaningful inferences. At the same time, it cannot be overly formal, or the system will become unusable.

Consider giving examples in high-level natural language, such as “implement selection just as in MacDraw but do not deselect other objects when an object is selected with a single click”. While this input method accommodates the user well, we currently seem to be far from being able to build machines that can interpret such input.

On the other hand, it is also easily possible to construct a demonstrational system that requires demonstrations of such complexity and formality that it is unusable for all practical purposes. That is, while it can draw powerful inferences in principle, the system is of little use in practice because its potential users are unable to provide it with correct demonstrations.[35]

We will now introduce the three demonstrational tools that we have built on top of the Elements, Events & Transitions model. Inference Bear (Section IV.1) uses *before* and *after* examples to describe *how* elements change in response to events. The Expression Finder (Section IV.2) uses *positive* and *negative* examples to describe *when* elements change in response to events. Grizzly Bear (Section IV.3) combines the functionality of Inference Bear and the Expression Finder, and is the most powerful tool.

This chapter makes extensive use of snapshots from our prototype implementation in C++, which runs on Sun SPARC stations running the Motif window manager on top of the X window system [Sche86]. The prototype implementation additionally makes use of an existing user interface builder, SX/Tools [Kueh92]. We have also implemented our own text editor based on the Motif text editing widget (so that we could seamlessly integrate a text editor with the rest of the system).

The SX/Tools interface builder lets user interface designers create new widgets by copying them from a palette, delete them via choosing “Cut” from their background menu, and modify them via a properties editor. An important feature is that it lets designers attach user-defined properties to widgets which can then also be edited with the properties editor. While these properties have no effect on the visual appearance of a widget we have found them useful for marking widgets

---

35. We have relied on usability testing to ensure that our tools are usable (Chapter VI), and on comparisons to previously built systems to ensure that our tools cover a unique and wide spectrum of situations (Section II.3).

for subsequent operations (e.g. through a boolean *IAmSelectable* property for widgets that can be selected) or for pointing to related widgets (e.g. through string-valued *from* and *to* properties for a line that connects two objects).

The user interface builder can also display the widget layout hierarchy as a tree structure, which provides an alternative way of selecting widgets (by clicking on their representation in the layout hierarchy rather than on the widgets themselves). This is crucial for demonstrating that an invisible widget becomes visible, as there would otherwise be no way of selecting the widget in order to edit its visibility property.

IV.1 Demonstrating How Elements Change

This section introduces Inference Bear, a demonstrational tool that can infer how elements change in response to an event. Inference Bear is a semi-acronym that refers to the tool's *before* and *after* input methodology ("**inference** from **b**efor**e** and **a**fte**r** snapshots"). With this technique, designers describe desired functionality in the following form: "*if* I am in this kind of situation (before snapshot) *and* the following event occurs (trigger event) *then* the user interface should look like this (after snapshot)"

One such pair of snapshots represents one *example* to Inference Bear. One *demonstration* may consist of one or more such examples. The result of a demonstration is the textual equivalent of the demonstrated behavior, in the form of an EET transition (see Section III.2.3).

IV.1.1 Inference Bear's User Interface

Figure 4-1 shows the control panel that appears when the user first invokes Inference Bear. Giving one example to Inference Bear consists of working through the four iconic buttons from left to right. (In our terminology, one *demonstration* consists of one or more *examples*.)

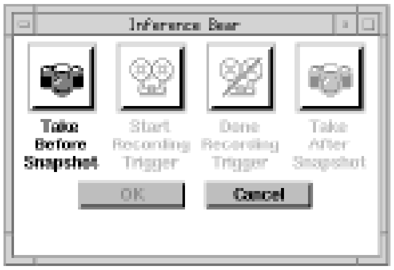


Figure 4-1: Inference Bear

The designer first sets up for the *before* snapshot by editing the current design just as she would in design mode, using the interface builder. She then clicks on the left-most button to tell the system that she is done editing for the *before* snapshot.

The next step is to tell the system which event triggers the behavior to be demonstrated. Inference Bear can deal with 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.

However, it also poses an interesting problem in recording the triggering event: there sometimes is no way of performing the triggering event without also causing extraneous events. For example, if the desired trigger event is the pressing of a mouse button over a certain 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 yet more mouse motion events. How will Inference Bear know which event is the one that triggers the behavior to be demonstrated?

There are several solutions to this problem. Peridot [Myer88] uses a "simulated mouse" which is an icon depicting a mouse and the state of its buttons. The user can then use the state and position of the simulated mouse to demonstrate mouse-dependent behavior, freeing the real mouse for meta-level commands. DEMO [Wolb91] always records a *press* event and then asks the user whether she really meant a *press* event, or whether she meant one of three other event types (*release*, *enter* and *leave* in our terminology). Marquise [Myer93] uses the keyboard to start and stop the recording. We use an alternative technique which uses time to distinguish the triggering event from the other events.

Figure 4-2 shows Inference Bear's control panel just after the "Take Before Snapshot" button has been clicked.

The "Take Before Snapshot" and "Take After Snapshot" buttons of Inference Bear's control panel are temporarily replaced with controls for recording the

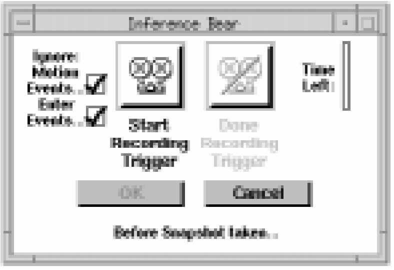


Figure 4-2: Capturing Events with Inference Bear

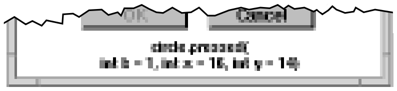
triggering event. When the user clicks on the “Start Recording Trigger” button, Inference Bear records user events for approximately five seconds and then uses the last event that was recorded before time ran out. The “Time Left” bar proportionately shrinks during these five seconds and Inference Bear beeps similar to a camera in “automatic timer mode” (five short beeps followed by one long beep). The “Ignore Motion Events” checkbox is best left checked unless one is actually recording a motion event.[36] The same is true for the “Ignore Enter Events” checkbox, if for no other reason than that the large number of enter and leave events is distracting.[37]

---

[36] Otherwise, recording e.g. a *press* event would require quite a steady hand, or one will cause additional *motion* events.

[37] In our prototype implementation, we encountered one situation where the “Ignore Enter/Leave Events” box *had* to be checked to correctly record an event, namely for recording menu selection events - the particular toolkit we used immediately sent a trailing *enter* event from the element under the mouse after pull-down or pop-up menus disappear.

Figure 4-3 shows part of the control panel after the event was recorded (for this particular event, *b* indicates that the left-most mouse button was pressed, *x* and *y* indicate the location of the click[38]). This feedback helps acquaint the designer with the types of events available.[39]



```
circle.pressed(
int b = 1, int x = 16, int y = 14)
```

Figure 4-3: Feedback on Recorded Events

This feedback is sometimes indispensable, for example for demonstrating that two text fields continuously display the numeric coordinates of the mouse cursor as the user moves the mouse over an application canvas (because the

---

[38] The actual events coming in from the user interface are particular to the user interface toolkit used.

[39] This feedback on events is at a very low level, which is desirable and undesirable at the same time. On one hand, the parameter type and name directly correspond to the parameters in the generated transition. On the other hand, the brief parameter names are hard to read. A refined version of Grizzly Bear used the compromise approach of displaying the actual events while recording and then showing a natural-language equivalent afterwards: “What should happen when ‘circle’ is ‘pressed’?” (see Section VI.1).

designer could otherwise not find out the motion event's precise *x* and *y* values for demonstrating this behavior).

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 issue a warning and prevent the designer from continuing when a mistake was made, such as not recording any event, or recording a different type of event than in previous examples of the same demonstration.

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 in this situation, the *after* state, and by pressing the "Take After Snapshot" button to let the system know that one is done with editing.

Inference Bear responds by running its inference engine on this example and by showing the resulting EET transition in the text editor. The interface design is reset to the state it was in before it was edited for the *before* snapshot.[40] The designer then tests if the inferred behavior is what she had in mind (by either

---

[40] It is interesting to note that our prototype implementation simply uses a different instantiation of the inference engine itself to implement the resetting. We take a snapshot before the design is edited for the *before* state and a snapshot after the editing for the *after* state, and then let this engine instantiation compute and execute a transition which transforms the interface back to the former state.

being able to read the text of the generated 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 permanently 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. She can also choose “Cancel” at any time to leave demonstration mode, which also removes the generated transition from the editor.

We will now provide several examples of Inference Bear’s use.

IV.1.2 Example: Popping Windows Revisited

We revisit the introductory modelling example of Section III.2.4 here, and show how a designer can construct it by demonstration. The functionality to be shown is that a Properties button will bring up a Properties window, and that the “OK” and “Cancel” buttons of the Properties window dismiss the window. This functionality can be defined in three demonstrations. We will discuss one of these three demonstrations in complete detail.

Figure 4-4 presents a detailed storyboard of the demonstration that tells Inference Bear that pressing the Properties button will disable this button[41] and

---

[41] The toolkit we used for this example does not provide an “enabled” attribute for buttons so that we have substituted changing their text color instead (white text on a gray background indicates a disabled button).

show the Properties window. (Please go through the storyboard of Figure 4-4 now. The generated EET transition is shown below.)

```
Transition PropertiesButton.pressed()
{
     PropertiesWindow.mapped := 1;
     self.textForeground := "white";
}
```

This introductory example presents a simple case in which a single example is sufficient to draw the desired inference. The designer can describe more complex behavior by giving additional examples.

IV.1.3 Example: The Moving Button

Assume 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 pressed. (No claim is made that this behavior represents good user interface design, of course. We use this example here only because it is a suitable minimal case of a multi-example demonstration.[42])

Figure 4-5 shows a condensed view of the three examples that are needed to demonstrate this behavior. We have superimposed the *before* and *after* snapshots so that the screen shot shows the *after* snapshot while the dotted shadow

---

[42] This particular example also shows that our design environment in no way enforces “good human-computer interface practice” but rather leaves all stylistic decisions to the interface designer.

Narration

1. The designer first sets up for the *before* snapshot by editing the interface design. In this particular case, she hides the Properties window (we show an outline here for the hidden window).

2. She lets Inference Bear know that she is done by clicking the Take Before Snapshot button.

3. She clicks the Start Recording Button. This switches the interface design from “edit mode” to “event recording mode” and starts the timer.

4. She holds the left mouse button pressed over the Properties button until Inference Bear’s event recording timer runs out. A small iconic cursor is shown which indicates the type and position of the click (borrowed from Marquise [Myer93]).

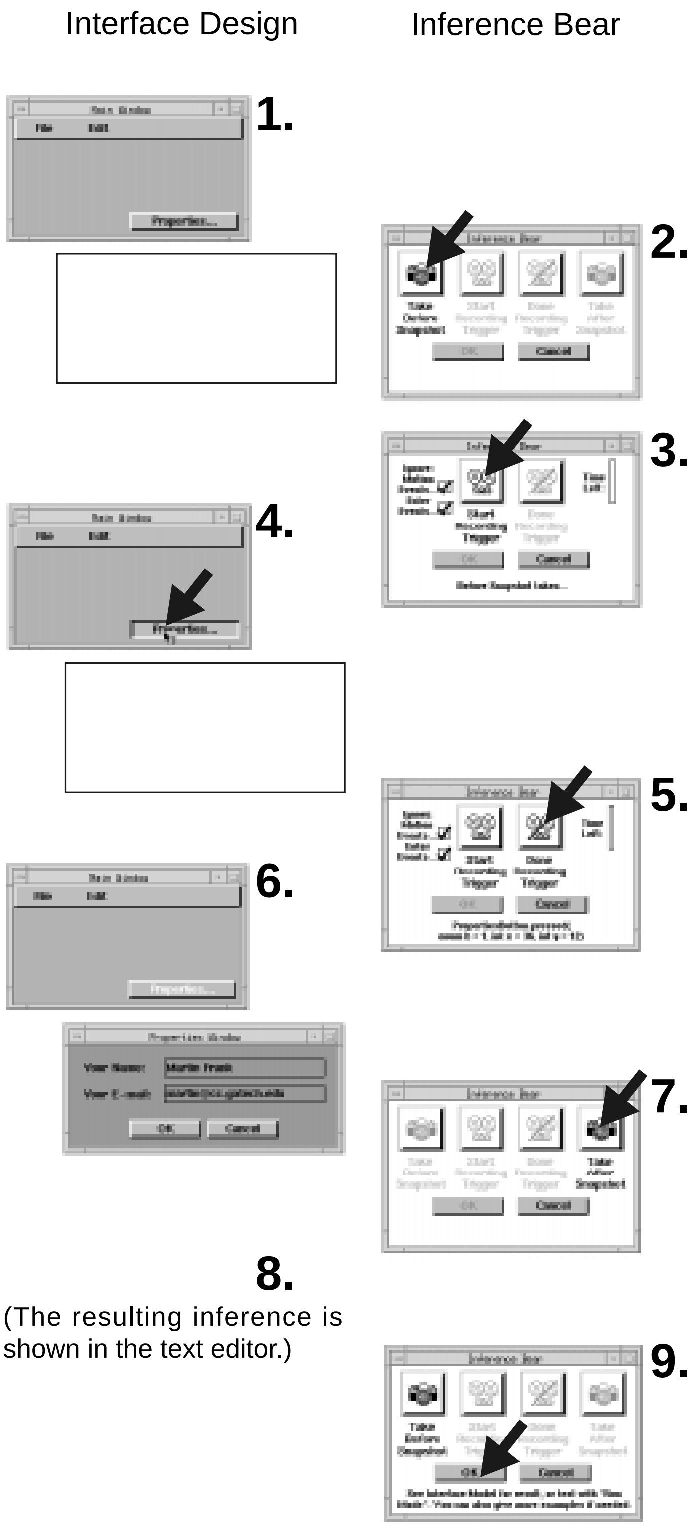
5. She clicks the Stop Recording Button which switches the interface design back into editing mode Inference Bear shows the recorded event in its status line.

6. She edits the interface design for the *after* snapshot by disabling the Properties button and by showing the Properties window.

7. She lets Inference Bear know that she is done by clicking the Take After Snapshot button.

8. Inference Bear shows the textual equivalent of its inference, and resets the interface design to the *before* state. The designer can now switch to run mode to verify the inference.

9. She OK’s the inference in this scenario.



Before Snapshot taken...

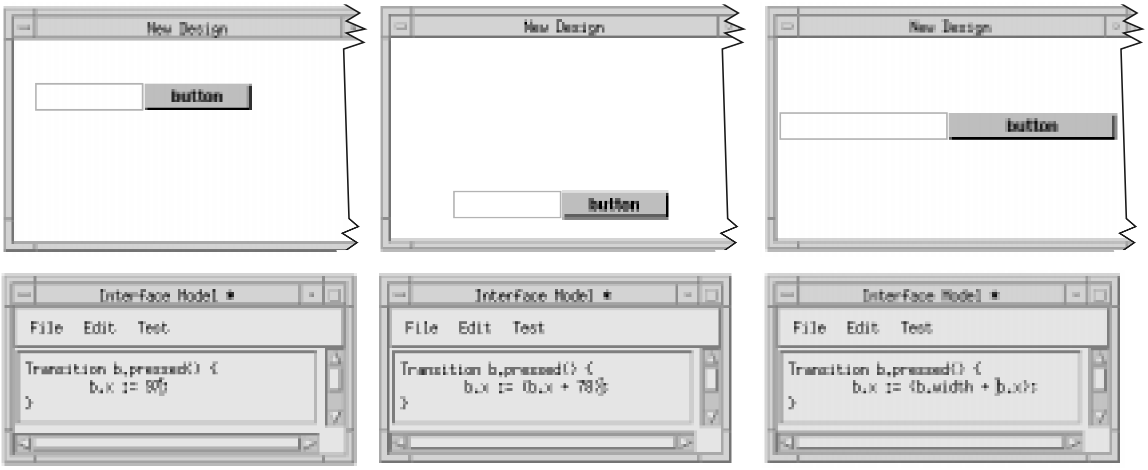
Your Name: Martin Frank

Your E-mail: martin@cc.gatech.edu

(The resulting inference is shown in the text editor.)

Figure 4-4: A Detailed Storyboard of Inference Bear

indicates where the button was located in the *before* snapshot. The recorded event is a click on the button in all three examples. Inference Bear responds with the scripts shown in the “Interface Model” screen shots after each example.



```
Transition b.pressed() {
        b.x := 89;
}
```

First Example.

```
Transition b.pressed() {
        b.x := (b.x + 78);
}
```

Second Example.

```
Transition b.pressed() {
        b.x := (b.width + b.x);
}
```

Third Example.

Figure 4-5: Inference Bear’s Successive Inference Refinement

After the first example, Inference Bear’s conjecture is that the button moves to the absolute position shown in the *after* snapshot, which is the simplest solution to this demonstration. Inference Bear always uses the simplest solution because there generally is an infinite number of more complex solutions[43], and because there is no way of choosing one solution over the other without using domain knowledge.[44]

---

[43] In this example, this demonstration could e.g. also be interpreted as “the button is horizontally centered within the window” or “the button moves to this absolute position only when the window background is white”.

The designer now gives the second example shown in the center column of Figure 4-5. She has moved the button to a different position for the before snapshot and has shown that the button again moves to the right (rather than to the same absolute position as in the first example). Inference Bear responds by refining the inference as shown. It has now inferred that the new button position is relative to the old one.

This solution will suffice if the width of the button never changes, but we will continue the demonstration here for the sake of completeness. So far, Inference Bear has inferred that the button moves by a fixed amount of pixels rather then by its own length. The general solution is found after the designer gives the third example shown to the right of Figure 4-5.

The number of examples required for an inference depends both on the computational complexity of the inference and on the quality of the examples. Poor examples are examples that are identical or nearly identical to previous examples. For example, had the designer demonstrated the first example of Figure 4-5 twice then Inference Bear would still have solved the demonstration using a constant. Ideal examples are examples that invalidate the current solution while being consistent with the desired solution. In Figure 4-5, the second exam-

---

44. We will later discuss why we have chosen not to make use of domain knowledge in our reasoning, and what the advantages and disadvantages are of not doing so (Section IV.5.2).

ple 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.[45] Section IV.4.2 explains Inference Bear's reasoning in detail.

IV.1.4 Example: Setting the Color of All Selected Elements

The previous two examples have dealt with changes to concrete, known objects. Inference Bear can also infer changes to a dynamic set of elements. A common example of such behavior is changing the appearance of the currently selected objects.

Figure 4-6 illustrates a demonstration of just such a case. The main window belongs to a small editor application that lets users create, select, and delete circles and ellipses. (We have used this example for our usability studies of Section VI.2, which is why the window is titled "Task 5".) The purpose of the new Color Palette window is to set the color of the currently selected objects to the current color of the Color Field when the Apply button is clicked. We assume that the

---

45. The need for good examples is not unique to Inference Bear - all programming by demonstration systems depend on the users' ability to provide good examples. For example, Henry Lieberman writes: "Teaching successfully depends, to a large extent, on the art of choosing good examples to present to students. (...) Good examples are ones that clearly illustrate the ideas that they are trying to convey, in the sense that knowing the idea makes a solution to the example possible." [in Cyph93, page 62].

designer has previously demonstrated that clicking on the Color Field makes it cycle through the colors red, blue, and green.[46]

Showing the functionality of the Apply button takes two examples, the first of which is shown in the upper half of Figure 4-6.

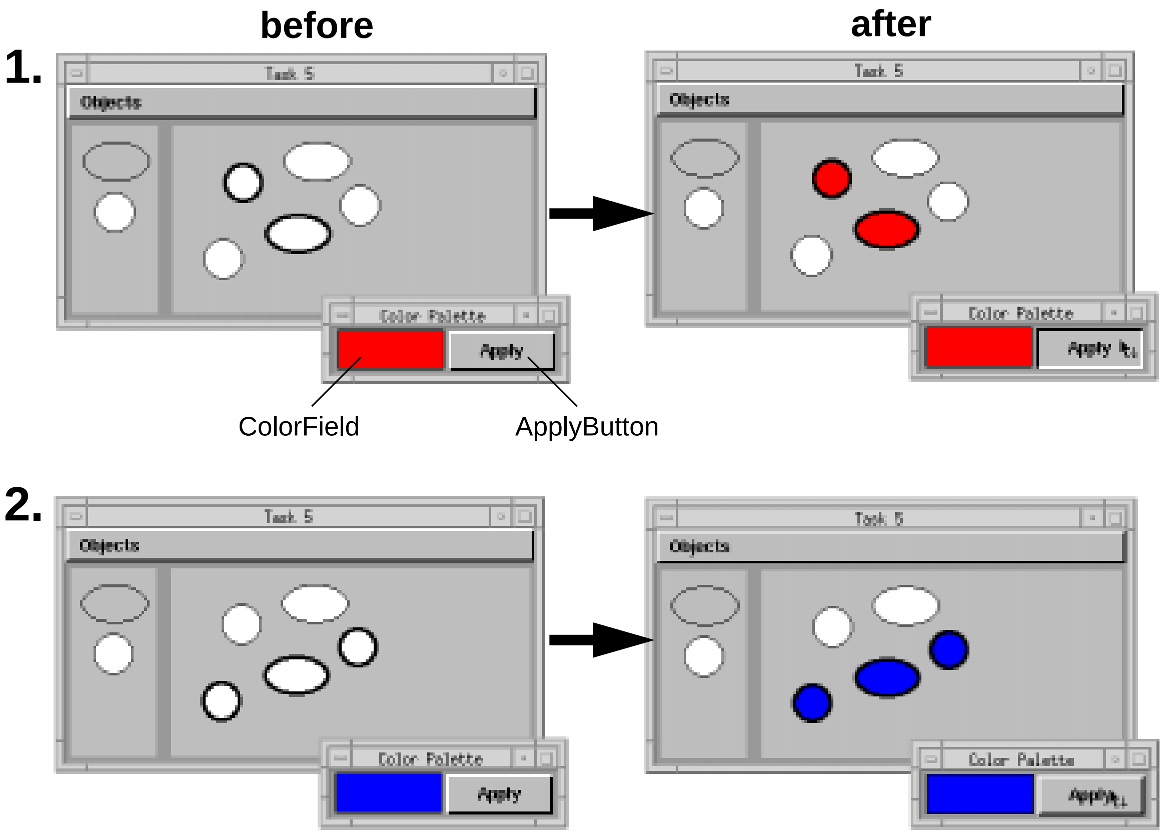


Figure 4-6: Setting the Color of All Selected Elements

The solution after the first example is shown below. As before, Inference Bear is conservative in its inferences and chooses the simplest solution, namely

---

46. We will show how designers can demonstrate this behavior in Section IV.3.2.

that these exact two elements change their color to red whenever the Apply button is pressed.[47]

```
Transition ApplyButton.pressed()
{
     auto2.fillForeground := "red";
     auto4.fillForeground := "red";
}
```

The desired solution is found after the second example, shown in the lower half of Figure 4-6.

```
Transition ApplyButton.pressed()
{
     (*.lineWidth==3).fillForeground :=
                           ColorField.fillForeground;
}
```

As before, the user can verify the behavior by interactive testing or by analysis of the textual inference.

IV.1.5 Example: Using Rubber-Banding to Create Lines

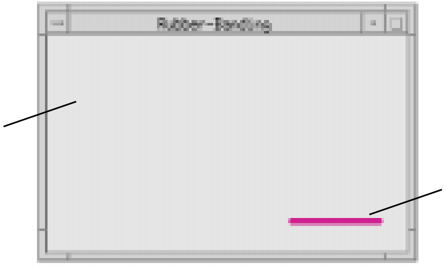
In this example, we will show how designers can construct a user interface in which lines are created using rubber-banding. That is, pressing the mouse but-

---

[47] Inference Bear could solve this demonstration after this initial example if it had a deeper understanding of what the designer is trying to do (if it had domain knowledge). We will later discuss why building domain knowledge into Inference Bear is nevertheless undesirable (Section IV.5.2).

ton sets one end-point of the line, and the other end-point then follows the mouse until the button is released.

This behavior can be demonstrated in its entirety. It requires three demonstrations - one each for the *press*, *release* and *motion* events associated with creating a line. The designer first builds the static user interface as shown in Figure 4-7.



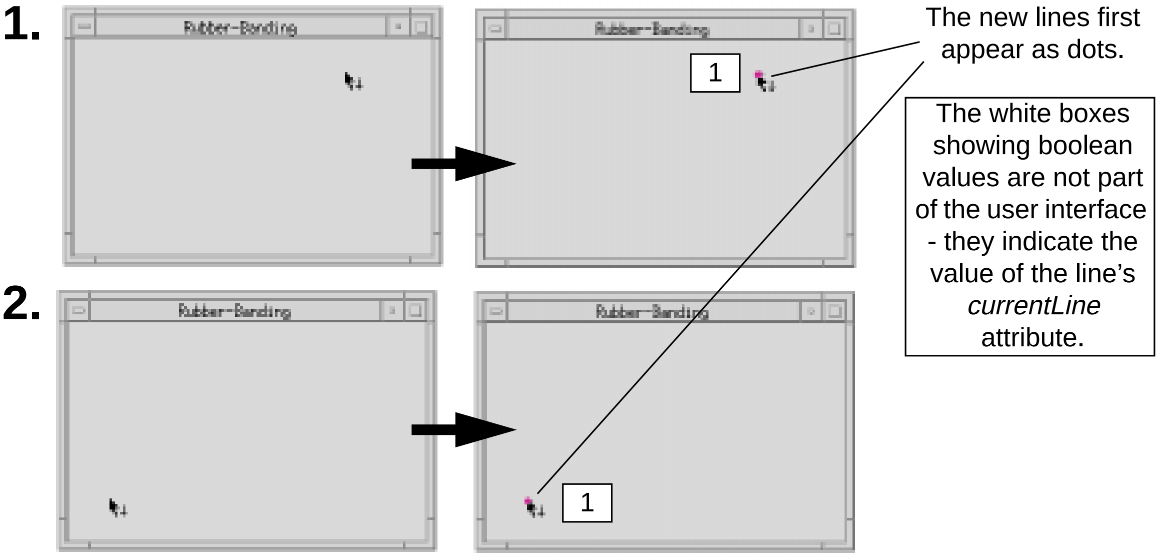
Canvas

Line
(This element will serve as a prototype for the lines to be created. The designer will normally hide it from end users by setting its "mapped" attribute to false.)

Figure 4-7: Static User Interface Layout for the Rubber-Banding Lines

She also attaches a new boolean attribute to the Line object called *currentLine*. This attribute will later make it easy to track which of the potentially many lines at run-time is the one which we just created. The prototype line is normally invisible to end users.

We are now ready to demonstrate behavior. Figure 4-8 shows the first demonstration, using two examples in which the designer shows that pressing on the canvas creates a new line, which first appears as a dot. Like in Marquise [Myer93], the designer can use the icon dropped by the *press* event to place the dot at the correct position. The *before* snapshots are again shown to the left, the *after* snapshots to the right. Inference Bear's response is shown below.



The new lines first appear as dots.

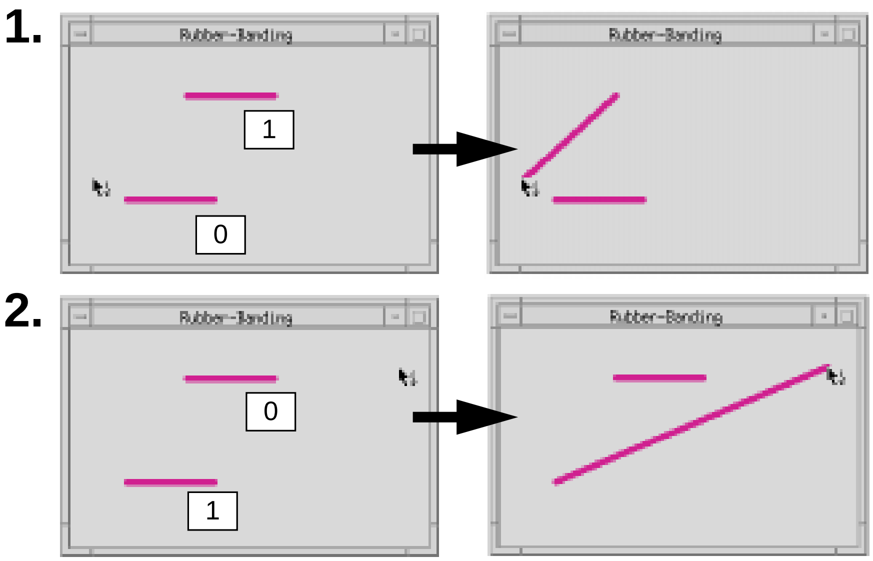
The white boxes showing boolean values are not part of the user interface - they indicate the value of the line's *currentLine* attribute.

Figure 4-8: Rubber-Banding Lines: Behavior of the *Press* Event

```
Transition Canvas.pressed(coord x, coord y)
{
        element n := create Line;
        n.mapped := 1;
        n.currentLine := 1;
        n.x0 := x;
        n.x1 := x;
        n.y0 := y;
        n.y1 := y;
}
```

We will now demonstrate how the line follows the mouse in Figure 4-9. The corresponding output is again shown below.

```
Transition Canvas.motion(coord x, coord y)
```



The value of a user-defined attribute can be edited using the interface builder's property editor, just like the value of a predefined attribute.

The designer edits the attribute here in order to demonstrate that only the line whose *currentLine* attribute is 1 (true) follows the mouse.

Figure 4-9: Rubber-Banding Lines: Behavior of the *Motion* Event

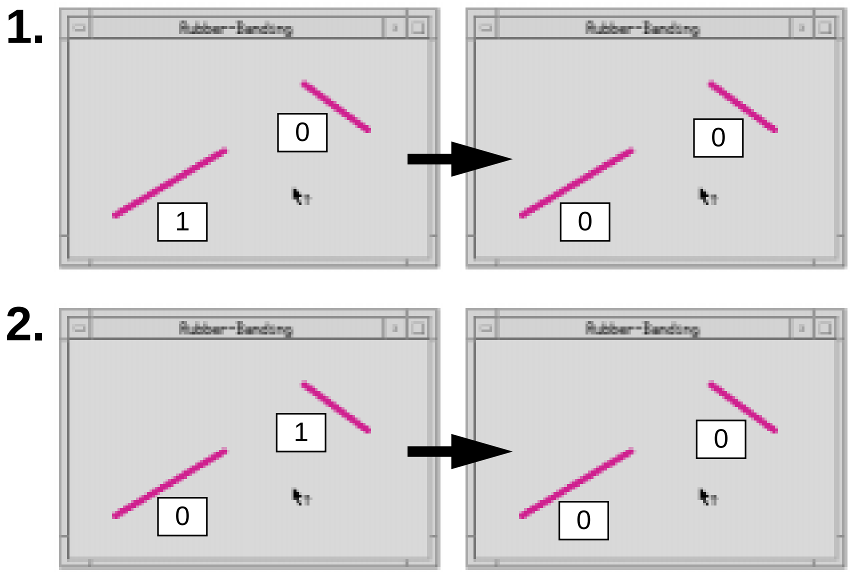
```
{
        (*.currentLine==1).x1 := x;
        (*.currentLine==1).y1 := y;
}
```

Finally, we will demonstrate that releasing the mouse button will set the *currentLine* attribute of the newly created line to false. (Otherwise, all lines will follow mouse motion events in the future.) Figure 4-10 shows the demonstration.

The listing below shows the resulting transition.

```
Transition Canvas.released()
{
        (*.currentLine==1).currentLine := 0;
}
```

With this third demonstration we have now completed the behavior for lines, and switching to Run Mode lets us create lines by pressing and dragging.



The super-imposed white boxes showing boolean values are again not part of the user interface - they indicate the value of the *currentLine* attribute in the snapshots.

What is being shown here is that the *currentLine* attribute of all lines is set to *false* when a mouse button is released over the canvas.

Figure 4-10: Rubber-Banding Lines: Behavior of the *Release* Event

Figure 4-11 shows a playful use of our skeletal graphical editor (it consists of eight lines).

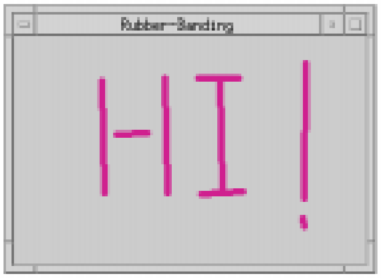


Figure 4-11: Rubber-Banding Lines: Playful Interaction with our Interface Design

The introduction of abstract attributes for graphical objects lets the designer create sophisticated interaction techniques. At the same time, dealing with abstract attributes requires a higher cognitive effort from the designer.

In this particular example, an alternative to introducing an abstract attribute is to mark the currently created line by temporarily changing its color. That is, we set the color of the line to, say, green when it is created at the *press* event, and set it again to the standard color for lines upon the *release* event. This way, we can tell what the newest line is based on its green color, and we can make one of its ends follow the *motion* events.[48]

IV.1.6 Example: Creating Objects by Dragging and Dropping

Finally, we will show how designers can demonstrate “drag-and-drop” interaction techniques to Inference Bear. Figure 4-12 shows an interface layout for a simple “file manager” application. It is named Mini-Finder after the well-known Apple Macintosh “Finder” application.

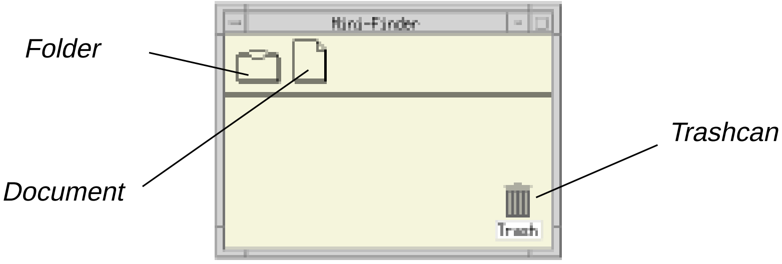


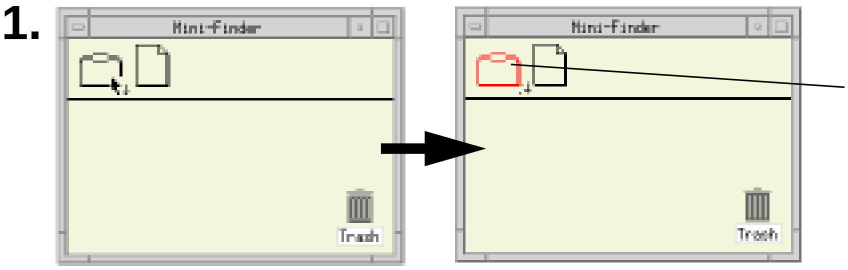
Figure 4-12: Mini-Finder: Initial User Interface Layout

---

[48] The difference between these two alternatives is visible to the end user. There is no alternative to the use of an abstract attribute if there is no difference in appearance at all between the currently-created line and other lines.

The designer makes all decisions about the details of the dragging and dropping. For this example, let's say that pressing the mouse down on the *Folder* icon makes a new red folder appear under the mouse which then follows the mouse until the mouse is released. Releasing the mouse button also gives the newly created element the standard black appearance.

Showing this functionality consists of one demonstration each for the *press*, *motion* and *release* events on the Folder icon.[49] The first demonstration shows Inference Bear that pressing down on the *Folder* icon makes a new red folder appear there (Figure 4-13, the red color appears grayish in the screen shots).



The new folder appears in red here, obscuring the original folder.

Figure 4-13: Mini-Finder: Demonstration for the *Press* Event

---

[49] It is worth noting here that all events between a *press* and a *release* event are received by the element which received the *press* event (the Folder icon here), even if the motion and release events occur outside the element's boundary. That is, the element monopolizes the event stream until the corresponding *release* event comes in. This technique is sometimes referred to as "screen grabbing".

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

So let us show that the newly created folder icon then follows the mouse. This is equivalent here with demonstrating that red objects follow the mouse (as the red color is what tells the newly created folder icon 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 one which is red follows the mouse. Figure 4-14 shows the two *before* and *after* snapshots of that demonstration.

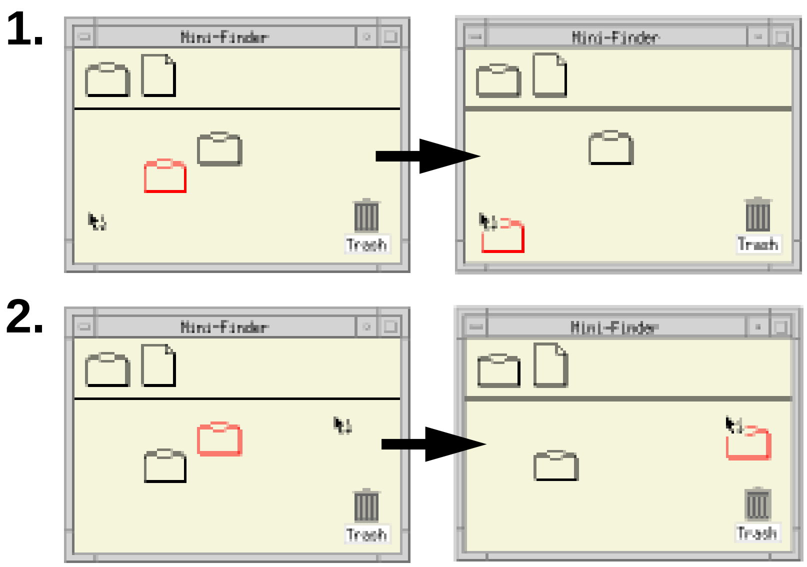


Figure 4-14: Mini-Finder: Demonstration for the *Motion* Events

If we test the current behavior now, we can create the first folder by dragging and dropping as desired, but it appears in red color on the canvas. If we create further new folders, we will drag *all* the previous folders as well as the new ones!

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. Similar to the previous demonstration, this is equivalent to demonstrating that all *red* objects change color upon such an event.[50] We again create two temporary objects to show this behavior (Figure 4-15).

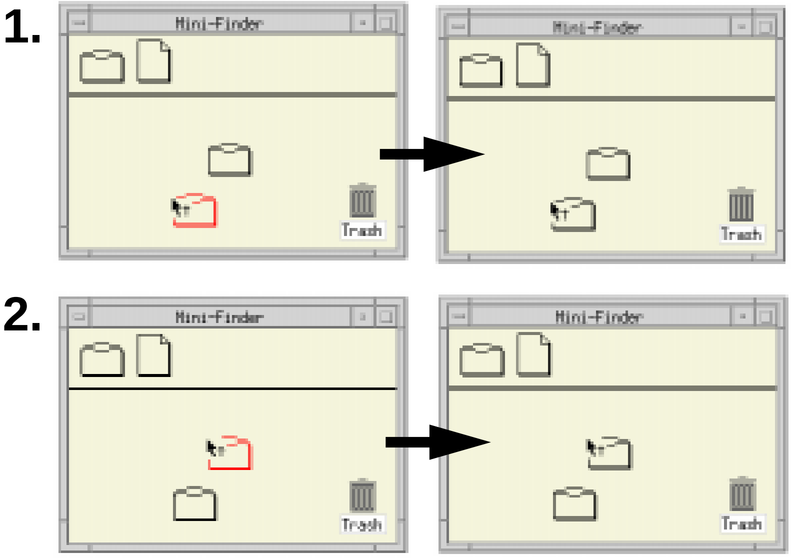


Figure 4-15: Mini-Finder: Demonstration for the *Release* Event

---

[50] We could also demonstrate that all red *folders* change color which will better guard against side effects if we later add other red objects to the user interface layout. We stick with the above solution above for its simplicity.

We have now fully specified the behavior for creating folders. The three transitions below correspond to the three demonstrations above, respectively.

```
Transition Folder.pressed()
{
        element n := create Folder;
        n.fillForeground := "red";
}

Transition Folder.motion(coord x, coord y)
{
        (*.fillForeground=="red").x := x;
        (*.fillForeground=="red").y := y;
}

Transition Folder.released()
{
        (*.fillForeground=="red").fillForeground
                                        := "black";
}
```

We could now demonstrate the behavior for creating documents in the same way, or we could textually copy the three transitions and adapt the copied transitions for documents (which is much faster and less repetitive but requires some understanding of the textual language). An even better solution is to generalize the generated transitions instead of simply duplicating them, so that they work for both folders and documents (which requires still more skills from the designer). We will later introduce Grizzly Bear, a more advanced demonstrational tool that can actually generate the generalized transitions (Section IV.3).

IV.2 Demonstrating When Elements Change

In the Elements, Events & Transitions model, the header of a transition describes which elements this transition applies to, and the circumstances under which it applies (e.g "this transition applies to objects on the main canvas when they are clicked, but only if we are in deletion mode"). The body of a transition then describes what happens (e.g "this transition deletes the element that matched the header").

Inference Bear specializes in generating sophisticated transition bodies. However, it always ties transitions to a particular element, and it always makes transitions unconditional. That is, Inference Bear can only generate transition headers of the simplest possible form (e.g. "this transition applies to object-6 when it is clicked").

This shortcoming severely limits the range of user interfaces that can be specified. For example, Inference Bear cannot generate a transition that will make *any* canvas object track the mouse. Other behavior that cannot be captured is conditionality such as "if I click on the canvas *and* I am in circle mode then...". In summary, designers could specify how elements change but they could not specify *when* that change occurs (other than tying it unconditionally to an event).

We initially addressed this shortcoming by building a companion tool for Inference Bear called the "Expression Finder". Its name refers to the tool's capability to find appropriate set expressions[51] based on the user's examples.

The Expression Finder uses the notion of *positive* and *negative* examples. Let us motivate why these are necessary in addition to the *before* and *after* examples that we introduced earlier. Assume we want to make a click on a canvas produce a circle, but only if we are in “circle mode”. Using the *before* and *after* technique we can describe that a circle appears where we click (these examples are implicitly “positive” examples in this terminology). However, we cannot express that this functionality is dependent on being in circle mode.

Using the *positive* and *negative* technique, we can express this conditionality by first going to circle mode and declaring the click a *positive* example, and by then leaving circle mode and declaring another click as a *negative* example. Expression Finder can then infer that the transition triggered by the click is additionally dependent on being in circle mode.

Expression Finder always leaves the body of the generated transition empty. Hence, transitions with complex invocation conditions are not entered via a single demonstration, they are rather entered via one demonstration to Inference Bear, one demonstration to the Expression Finder, and via copying the body of Inference Bear’s transition into Expression Finder’s transition.[52]

Having to give two demonstrations for one transition is both desirable and undesirable. On one hand, it keeps the size of individual demonstrations small.

---

[51] Sections III.3.1 and III.3.2 introduce set expressions and their use in transition headers.

On the other hand, it requires some familiarity with the textual language to cut and paste the output of the demonstrations. We will later present a tool which can indeed generate transitions with complex invocation conditions from a single demonstration (Section IV.3), and we will present quantitative data on how its usability compares with using the separate demonstrational tools (Chapter VI).

IV.2.1 Expression Finder's User Interface

Expression Finder is similar in appearance to Inference Bear. Each example to the Expression Finder consist of a snapshot of an interface state, a triggering event, and an example classification (Figure 4-16).

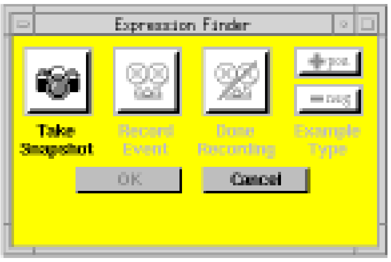


Figure 4-16: Expression Finder

The Expression Finder uses the same event recording mechanism as Inference Bear so that we will not discuss it again. There is no need for an *after*

---

52. Alternatively, the designer could highlight the transition generated by Inference Bear before demonstrating to Expression Finder. The system would then automatically merge the two transitions.

snapshot in an example as the Expression Finder is only concerned with *when* something happens. It is not concerned with what this “something” is - the transition body must later be filled in by other means.

Expression Finder allows the designer to interactively test the current behavior after each example, similar to Inference Bear. But how does the designer tell if the new transition gets triggered if its body is empty? Our solution was to make the system beep if an event in test-drive mode matched the expression of a transition with an empty body.[53] We will now present several examples of using the Expression Finder.

IV.2.2 Example: Making a Transition Dependent on the Context

We introduce the Expression Finder with a storyboard that shows how to generate a transition that depends on the current user interface mode. (Please go through the storyboard in Figure 4-17 now. Note that “MacDraw II 1.1” in the figure is actually an EET-based application that mimics Apple’s MacDraw.)

The textual output of this demonstration is as follows.

```
Transition ((*.name=="canvas") &&
             (PaletteLine.foreground=="black")).pressed()
{
}
```

---

[53] This solution is of course not ideal. We later addressed this problem better by combining the reasoning for the header and body of a transition (Grizzly Bear).

Narration

1. The designer edits the user interface design for the first example. In this particular case, we set up for a positive example by highlighting the line icon in the palette.

2. She tells the Expression Finder that she is done editing by clicking the “Take Snapshot” icon.

3. She starts the event recording.

4. She causes the triggering event, a mouse-down on the canvas.

5. She signals that she is satisfied with the recorded event.

6. She declares this first example as a positive example, meaning that something should happen in response to the mouse-down in this situation.

7. She edits the user interface for the second example by changing the color of the line icon in the palette.

8.-11. She again records a mouse-down event on the canvas. These steps are the same as steps 2.-5. above, and are not shown again.

12. She declares the second example to be a negative example - she does not want the behavior to be triggered when the line icon is not highlighted.

13. She accepts the inference in this scenario.

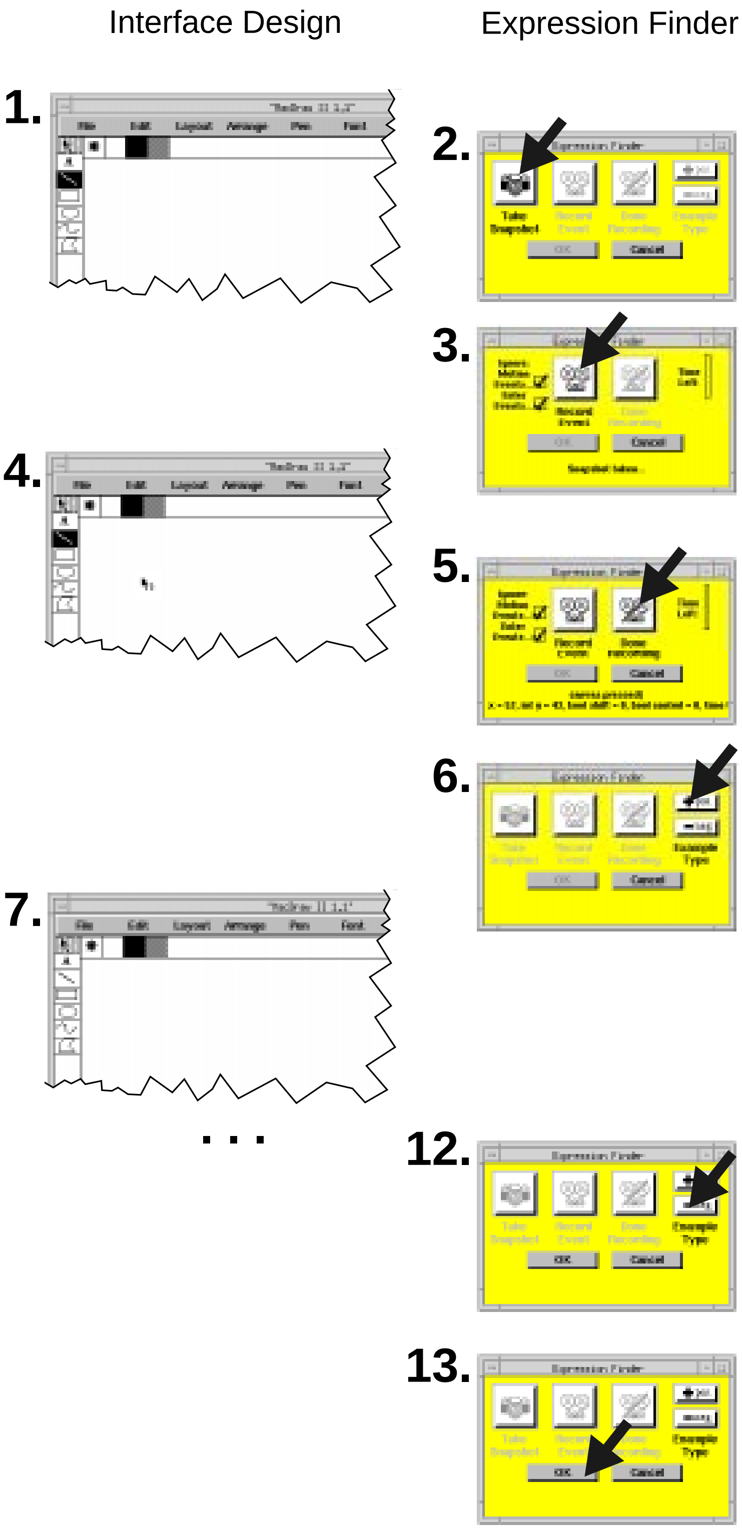


Figure 4-17: A Detailed Storyboard of Expression Finder

The transition is invoked when a *press* event occurs on an element named canvas if it also happens to be the case that the foreground color of the Palette-Line element is black. The transition body is empty so far. In this particular example, we can then use Inference Bear to generate a transition body which will make a new line appear (as described earlier in Section IV.1.5).

IV.2.3 Example: Defining A Transition for a Class of Elements

The Expression Finder can also be used to specify behavior that is exhibited by a class of elements. Consider the Mini-Finder application that we introduced earlier (Figure 4-12). Part of the functionality we want to specify is that all folders and documents can be dragged to the trash can. However, we of course do not want users to be able to accidentally drag the *palette* document or folder to the trash can! We can demonstrate for which elements this behavior applies by giving positive and negative examples.

We call the Expression Finder, and create some documents and folders on the canvas that will serve as positive examples (Figure 4-18). These new objects receive automatically generated names as shown because we did not explicitly name them.

For this particular example, let us first demonstrate for each of the four canvas objects that dragging them onto the trash can is a positive example. After giving these four positive examples, the Expression Finder’s solution is shown below.

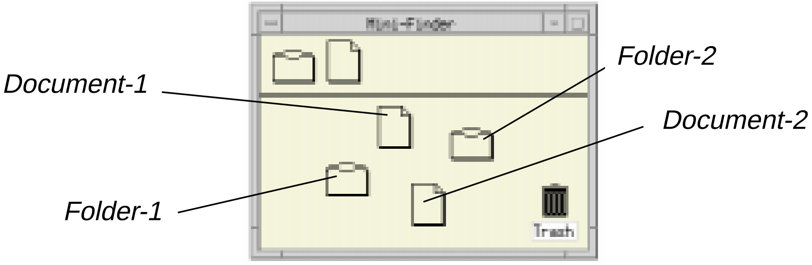


Figure 4-18: The Mini-Finder Revisited

```
Transition ((*.name=="Folder-1") ||
            (*.name=="Document-1") ||
            (*.name=="Document-2") ||
            (*.name=="Folder-2")).released()
{
}
```

As is apparent from the output above, the Expression Finder does not make an attempt to generalize until at least one negative example is given. This is because without a negative example there is no domain-independent way of telling which objects the behavior does *not* apply to.

Let us thus now show that the palette Folder and palette Document and the Trashcan itself can never be deleted by dragging them to the trashcan. This is again done by recording a release event over the trashcan that started with a press event over the respective objects. The Expression Finder then finds a minimal expression that tells the positive examples from the negative example, as shown below.

```
Transition ((*.imageName!="Trashcon.icon") &&
            (*.y>=32)).released()
{
}
```

In this particular case, the Expression Finder has concluded that the relevant objects are the objects on the lower part of the canvas that do not display the trashcan icon.

This is not necessarily the “best” solution to this demonstration. There is often a trade-off between the quality of Expression Finder’s solution and the number of examples that must be given. The above solution is good enough for constructing quick application prototypes (e.g. for early usability testing). The Expression Finder can indeed find the optimal solution here, namely “objects that are Folders or Documents but that are not the prototype Folder or the prototype Document”. However, this may take a number of examples (about twelve here) for this solution to be found because the *optimal* solution has to become the *minimal* solution. As mentioned earlier, there is no way of telling “optimal” from “minimal” without using domain knowledge.

The above transition correctly tells us which objects can be deleted. However, we have so far specified that behavior for these objects is triggered when they are released over *any* object, even though we have always shown that the objects get released over the trash can.

In the particular event system on which we built our prototype implementation, a *release* is received by the object that also received the corresponding *press* event, and the name of the object over which the *release* event occurred is sent as a parameter. As this parameter never changed between any examples the

Expression Finder assumes it is irrelevant. (We will discuss the importance of the event structure in Section VII.1.2.)

Given the particular event structure of our prototype system, we can complete the demonstration by giving two negative examples of release events onto objects other than the trash can. The final inference is shown below.

```
Transition ((*.imageName!="Trashcan.icon") &&
            (*.y>=32) &&
            (o=="Trashcan")).released(string o)
{
}
```

The inferred transition is now additionally dependent on the release event receiver (“o”) being the trash can.

As with all transitions inferred by the Expression Finder the body of the transition is to be filled in later either by hand or by using Inference Bear. In this example, the statement that implements deleting the dragged element is “delete this”.

Section IV.4.3 explains the Expression Finder’s internal reasoning.

IV.3 Demonstrating Both How and When Elements Change

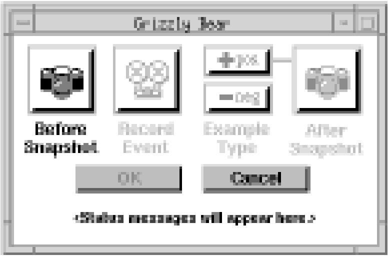
The previous two sections introduced Inference Bear and the Expression Finder. There are some advantages to using two separate tools, such as keeping the size of individual demonstrations small.

However, we observed in our usability testing that users are often confused about which of the two tools to call. We have also observed that getting started with the Expression Finder is difficult at first because it does not produce immediately observable new behavior. Finally, we wanted to push the limit of what can be achieved by demonstration alone as much as possible. (More precisely, our interest was in what can be achieved by demonstration alone if no knowledge is built into the tool about what is being demonstrated, as discussed in Section IV.5.2).

Grizzly Bear addresses these issues by combining the reasoning power of the previous two tools into a single control panel. Its name stems from simply being the bigger brother of Inference Bear.

IV.3.1 Grizzly Bear’s User Interface

Grizzly Bear’s user interface is shown in Figure 4-19. A single example is again given by working through the buttons from left to right. Each example is classified as positive or negative using one of the two “Example Type” buttons. Pressing one of these buttons implies that we are done recording, so that a separate “Done Recording” button is no longer needed.



<Status messages will appear here.>

Figure 4-19: Grizzly Bear

A *positive* example requires an *after* snapshot to show what happens in response to the event, as visually indicated by the line connecting the “pos” button to the “After Snapshot” button. Giving a *negative* example implies that no behavior will be triggered by the event in the context of the *before* snapshot. Hence, there is no need to explicitly show that nothing happens by giving an unchanged *after* snapshot for each negative example.

Grizzly Bear degenerates into Inference Bear if the user only gives positive examples. Grizzly Bear degenerates into the Expression Finder if the user never makes changes for the *after* snapshot.

IV.3.2 Example: Cycling Through Colors

An example of simple behavior that could not be demonstrated to Inference Bear is cyclically changing an attribute. For example, we mentioned a color field that cycles through colors every time it is clicked in Section IV.1.4, but we did not elaborate on how to demonstrate this behavior. Figure 4-20 re-introduces the color field example.

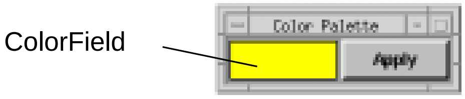


Figure 4-20: Cycling Through Colors

The behavior to be demonstrated is that, upon a *press* event, the color field becomes red if it is currently yellow. It becomes blue if it is currently red, and it becomes yellow again if it is currently blue. A detailed storyboard on how to demonstrate this behavior to Grizzly Bear is shown in Figure 4-21. (Please go through the storyboard now.) The output of this demonstration is shown below.

```
Transition ((*.name==”ColorField”) &&
     (ColorField.fillForeground==”yellow”)).pressed()
{
     self.fillForeground := “red”;
}
```

The storyboard shows that the color field becomes red if it is currently yellow. Analogous demonstrations can be given for the two other cases if all of the behavior is to be demonstrated, or the designer can textually copy the generated transition and change the color references.

We use the three-color scenario here because it makes for a good minimal example involving conditionality in a transition. No claim is made that cycling through colors is an adequate interaction technique for choosing colors on, say, a 24-bit deep screen. Designers can also use Grizzly Bear to let users choose from a color palette, or to simply let them type the color’s name.[54] Finally, if the color is an integral part not only of the user interface but of the application itself then custom application code will have to be tied to the user interface (e.g. in a desktop publishing application).

IV.3.3 Example: Making Some Objects Draggable but not Others

Section IV.1.6 introduced the Mini-Finder application when we showed how Inference Bear can be used to implement a “drag-and-drop” interaction technique. We later used Expression Finder to describe which elements can and cannot be deleted by dragging them to the trashcan (Section IV.2.3). Figure 4-22 again shows the layout of this application.

Grizzly Bear could also have inferred the behavior shown to Inference Bear and the Expression Finder, and it could have produced the transition body for the trashcan-dragging, not just the transition header. We will complete our discussion of the Mini-Finder application by explaining how Grizzly Bear can be used to make objects on the canvas draggable. Part of the challenge is to make sure that *only* the desired objects can be dragged but not others. In Section IV.2.3, the Expression Finder used an approximate solution that distinguished between the prototype objects and the canvas objects by using their relative vertical position.

A better solution is to attach a user-defined attribute to the objects in question, which makes it easier for the demonstrational tool to distinguish between

---

54. Additionally, an extended version of Grizzly Bear could infer that the color is to be taken from a list of colors (which “wraps around” when the end of the list is reached).

Narration

1. + 2. The designer takes a *before* snapshot of the color field being yellow

3. + 4. She records a *press* event on that field.

5. She declares the example to be positive, meaning that the behavior to be shown *will* be triggered if the recorded event occurs in a situation similar to the *before* snapshot.

6. + 7. She takes an after snapshot that shows that the color fields becomes red.
Grizzly Bear now displays its initial inference (which is that the color field *always* becomes red when it is clicked).

8. + 9. The designer takes a before snapshot which shows the color field being any color but yellow. We used red here.

10. + 11. She again records a press event on the color field. (These two steps are not shown as they are identical to steps three and four.)

12. She declares the example to be negative, meaning that the behavior will *not* be triggered if the interface is in the context of the *before* snapshot.
Grizzly Bear now displays its refined inference which is that the color field only becomes red is it has previously been yellow.

13. The designer OK's this inference in this scenario.

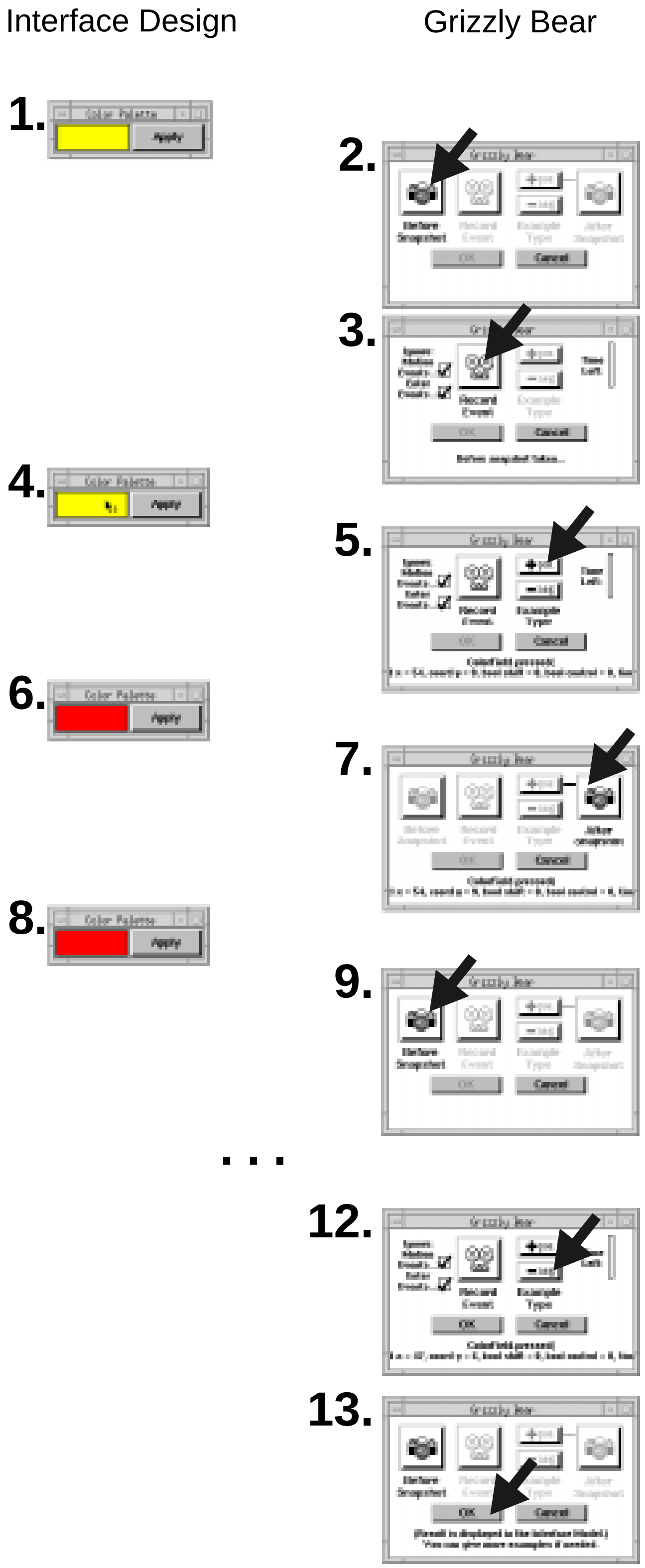


Figure 4-21: A Detailed Storyboard of Grizzly Bear

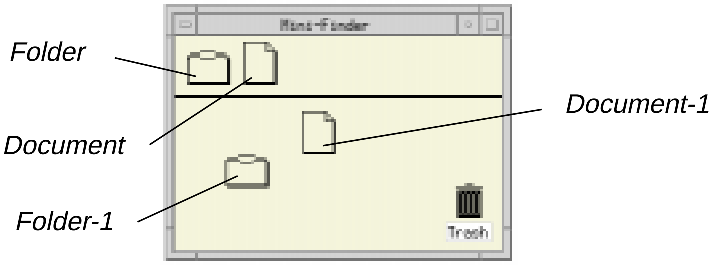


Figure 4-22: Mini-Finder

them, and which also produces a higher-quality solution. In this implementation of mini-finder's functionality we have attached an "IAmDraggable" attribute to the *Folder* and the *Document*. The default value of this string-valued attribute is "no". We have then shown how new objects are created by dragging (as in Section IV.1.6), but we have additionally shown that their "IAmDraggable" attribute is set to "yes" when they are created. This gives Grizzly Bear a more symbolic way of describing the objects that can be dragged.

We now describe in detail how the designer shows which objects can be moved on the canvas. Figure 4-23 depicts the first example, where we show that a move event originating on Document-1 results in the document moving there. We superimposed the dotted line over the screen shots to illustrate from which object the move event originated. The *before* snapshot is shown on the left, the corresponding *after* snapshot on the right of the figures. We indicated if the designer classified the example as positive or negative using their mathematical symbols.

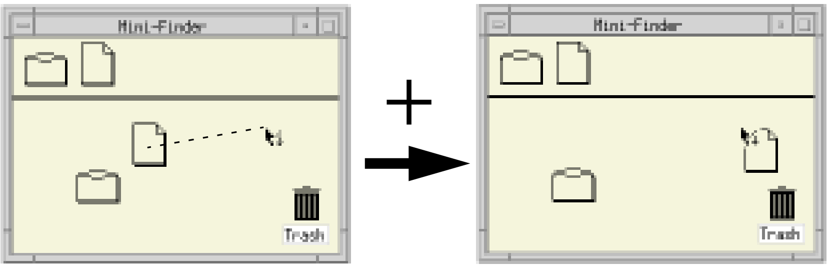


Figure 4-23: Moving by Dragging: First Example

After this first example, Grizzly Bear’s inference is that any move event on Document-1 will result in it moving to the absolute position shown in the *after* snapshot. (This is because Grizzly Bear knows nothing about the *concept* of moving - it could otherwise draw much better inferences from single examples. There are, however, also many advantages to not building in such concepts.)

The designer now gives the second example (Figure 4-24). It shows that a move event on Folder-1 will result in that object following the move event. Grizzly Bear’s inference now is that either Document-1 or Folder-1 can be moved that way.

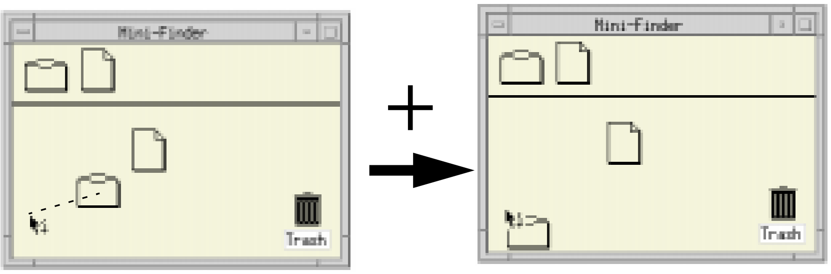


Figure 4-24: Moving by Dragging: Second Example

Grizzly Bear does not attempt to generalize to objects other than the ones actually used until at least one negative example is also given. (It will use its status line to ask the designer for an example of an object *not* exhibiting the demonstrated behavior if an example contains positive examples for more than one object.)

Figure 4-25 shows the final example, which is negative. It expresses that nothing happens when the Folder object is moved. As with all negative examples, no *after* snapshot is required.

Grizzly Bear now draws the desired inference, which is that any folder that has an “IAmDraggable” attribute that reads “yes” can be moved in that manner.

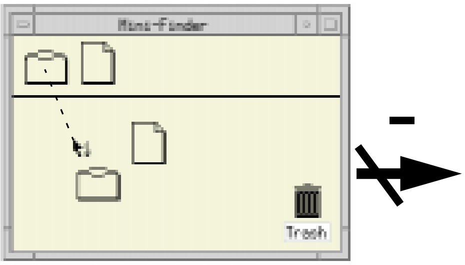


Figure 4-25: Moving by Dragging: Third Example

The actual text of the inference is shown below.

```
Transition (*.IAmDraggable=="yes").motion(
        coord x, coord y)
{
        self.x := x;
        self.y := y;
}
```

It is worth noting that it is rather by chance that we need three examples here because there are several minimal ways of telling the objects in the positive and negative examples apart. ("Minimal" here means using the smallest number of comparisons in the expression.) The minimal solution that was used is "*.IAmDraggable==yes". Another solution is "*.name!=Folder", telling the objects apart by their name. Grizzly Bear has no way of telling which of those two solutions is "better", as it has no semantic knowledge of what is being demonstrated. It chose the former solution here simply because of its lower alphabetic order (see Section IV.4.1.3). If Grizzly Bear would arbitrate between otherwise equivalent solutions by, say, reverse alphabetic order, another negative example (on the Document object) would be needed for this particular example.

IV.3.4 Example: A Complete Mini-Editor

Finally, let us show how a small editor can be built completely by demonstration. This application uses a different way of creating and deleting objects than the Mini-Finder application, and it also shows how object selection can be implemented, and how Grizzly Bear deals with menus. Another reason for presenting this application here is that we have used it for usability testing (Chapter VI). The initial layout is shown in Figure 4-26.

It can be used to create circles and ellipses, select them, and delete them. In more detail, the ellipse and the circle in Figure 4-26 serve as palette items: clicking on the ellipse will make the interface go into "ellipse mode", clicking on the

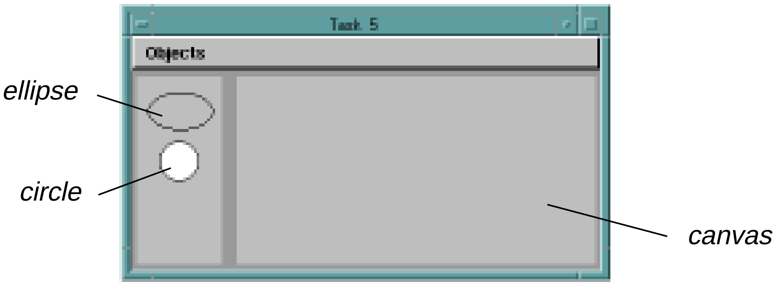


Figure 4-26: The Initial Layout of the Mini-Editor

circle will make it go into “circle mode” (we are currently in circle mode). Clicking on the canvas will produce either a circle or an ellipse there depending on the current mode (the new objects appear in white, just as the prototype object). These objects can then be selected by clicking on them, which is indicated by a thicker border. Finally, selecting “Delete” from the “Objects” menu will delete all currently selected objects. Figure 4-27 shows a snapshot of the finished mini-editor in use.

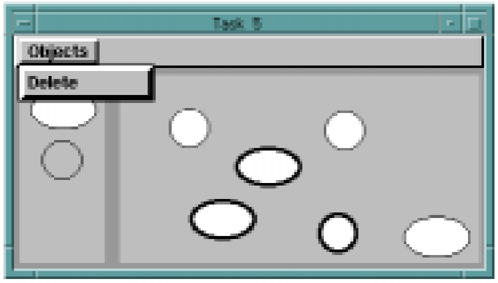


Figure 4-27: The Completed Mini-Editor in Action

In our usability testing, we subdivided the task of specifying this functionality into four sub-tasks which we will describe here in detail.

Task 5a was to implement the toggling of the circle and the ellipse in the palette. This can be done in two separate demonstrations, one each for the circle and the ellipse.

```
Transition circle.pressed()
{
        self.fillForeground := "white";
        ellipse.fillForeground := "gray";
}

Transition ellipse.pressed()
{
        self.fillForeground := "white";
        circle.fillForeground := "gray";
}
```

It can also be done in a single demonstration, using two examples. In this case, the output is as follows. This solution is indeed correct, as the two assignments are “conceptually executed in parallel” (see Section III.5.1).

```
Transition (*.name=="circle" ||
            *.name=="ellipse").pressed()
{
      ellipse.fillForeground := circle.fillForeground;
      circle.fillForeground := ellipse.fillForeground;
}
```

We can now toggle between circle mode and ellipse mode. Task 5-b was to implement the creation of objects in response to a click on the canvas, which also requires one demonstrations each for circle mode and for ellipse mode. Each of these demonstrations requires at least two positive examples for Grizzly Bear to infer the location of the new object, and at least one negative example for it to infer that the behavior is mode-dependent. The inferences are shown below.

```
Transition ((*.name=="canvas") &&
            (circle.fillForeground=="white")).pressed(
        coord x, coord y)
{
        element n := create circle;
        n.x := x;
        n.y := y;
}

Transition ((*.name=="canvas") &&
            (ellipse.fillForeground=="white")).pressed(
        coord x, coord y)
{
        element n := create ellipse;
        n.x := x;
        n.y := y;
}
```

We can now create ellipses and circles on the canvas. Task 5-c was to make the new objects selectable. The selection status was to be indicated via an object's border line width. This behavior minimally requires two positive examples and one negative example. In this case, Grizzly Bear can come up with a solution that tells the objects in the palette from the canvas objects by their relative horizontal position. A higher-quality solution like the one shown below requires approximately ten examples.

```
Transition (((*.class=="Circle") ||
             (*.class=="Ellipse")) &&
            (!((*.name=="circle") ||
               (*.name=="ellipse")))).pressed()
{
        self.lineWidth := 3;
}
```

An alternative solution is to mark the selectable objects with an extra attribute such as “IAmSelectable” (the same technique was used earlier in

Section IV.3.3). In this case, the solution can be demonstrated with one positive and one negative example.

Finally, task 5-d was to make the “Delete” menu item delete all currently selected objects. This behavior requires only two positive examples each of which has shows the deletion of a different set of selected objects. The output appears below.

```
Transition SXMenu_1.menuInvoked(
        enum menu == "Delete")
{
        delete (*.lineWidth==3);
}
```

We can now create and delete objects as originally intended (Figure 4-27). We chose this example for our usability testing because it takes less than an hour to construct while still containing challenging demonstrations (object creation and deleting, functionality exhibited by sets of objects).

No claim is made that this editor is complete, of course. For example, users cannot even deselect objects. While this behavior can easily be demonstrated (same demonstrational complexity as selecting objects) there was no benefit in having subjects perform repetitive demonstrations.

Other desirable behavior that can be entirely demonstrated to Grizzly Bear includes making canvas object movable (see Section IV.3.3), providing a “Deselect All” menu entry (a two-example demonstration), and providing a floating color palette (see Section IV.1.4).

IV.4 The Inferencing Mechanism

This section explains the internal reasoning process of the demonstrational tools. We will first describe the principles underlying our inferencing mechanism, as this will provide context for the design decisions presented later.

It contains no domain knowledge.[55] The design goal of our demonstrational tools is that they be useful for a broad range of application domains. Consequently, we cannot base their inferencing on knowledge about a particular domain. A disadvantage of this domain-independent[56] approach is that we cannot make use of such knowledge to aid the inference process. On the other hand, the tools can be used for many domains. It can potentially also be used at any level of abstraction. For example, it can be used to demonstrate how a dragged object follows the mouse pointer, but it can also be used to demonstrate that the number of employees increases by one if a new employee is hired.[57]

---

[55] The reasoning mechanism proper contains no domain knowledge, but it contains “hooks” (function calls) which one can use to influence the reasoning process based on domain knowledge. Thus, it is fair to say that the overall reasoning contains “little” domain knowledge.

[56] No reasoning is completely independent of its domain, of course, but we lack a better name.

[57] Being able to demonstrate at an abstract level is dependent on having an editable visualization of those abstractions, of course.

It finds the simplest possible solution. That is, the inference mechanism produces the transition with the fewest statements and the least complex invocation expression. The mechanism cannot otherwise tell which of two solutions is superior because of its lack of domain knowledge.

It can infer changes to any attribute. The engine can reason about any attribute of an EET element. It can infer assignment of a constant (a.color := "blue") and assignment from another variable (a.color := b.color) even if it does not know about the type of the attribute. It can make more advanced type-specific inferences if it does (such as "a.x := a.x + 1/2 * a.width").

The above statement should not be confused with "it can infer changes to any attribute *of an arbitrarily complex nature*" - all (successful) domain-independent demonstrational engines look for simple relationships[58].

IV.4.1 Enabling Technology Used by All Tools

This section will describe concepts and algorithms that are used for the reasoning of Inference Bear, Expression Finder and Grizzly Bear. We will first introduce source and target variables. Simply stated, source variables are potential parameters of a solution while target variables are those that must be solved.[59]

---

58. Some early machine learning efforts in the artificial intelligence field failed precisely because they tried to infer arbitrary programs from examples of input and output.

IV.4.1.1 Source and Target Variables

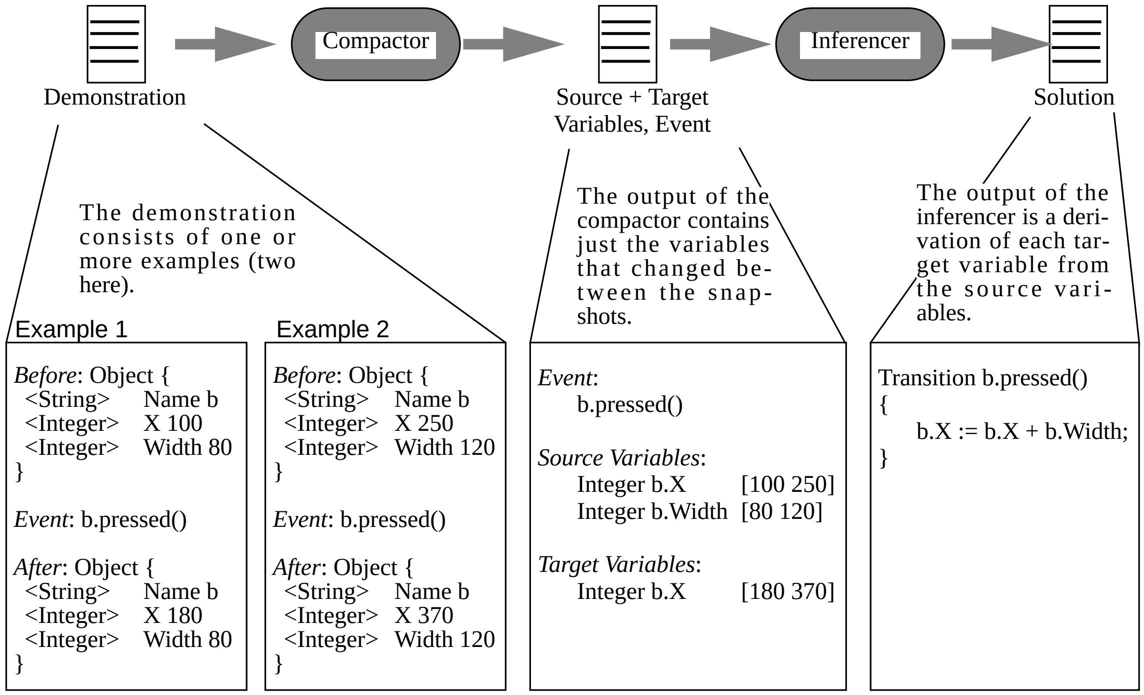
Figure 4-28 shows a small but complete example of Inference Bear's reasoning process that will introduce the key concepts of *source* and *target* variables. The user has given two examples of a button moving one button length to its right in response to clicking on it (the same example that was described from the user's view in Section IV.1.3). Again, no claim is made that this is particularly desirable behavior, but it provides for a minimal yet complete example of the reasoning process.

The inferencing is done in two stages. In the first phase, the "Compactor" reduces the amount of objects and attributes that the inferencing process has to be concerned with. This is done by eliminating all objects and attributes which remain constant in the examples. The result of the Compactor is a list of "source" variables and a list of "target" variables.

Source variables are potential parameters of a solution. Attributes become source variables if they change between any two Before snapshots.

---

59. Note that while source variables are relevant for all demonstrational tools, target variables are only used by Inference Bear and Grizzly Bear. We present both source and target variables here because their discussion naturally belongs together.



The demonstration consists of one or more examples (two here).

The output of the compactor contains just the variables that changed between the snapshots.

The output of the inferencer is a derivation of each target variable from the source variables.

```
Before: Object {
 <String>     Name b
 <Integer>    X 100
 <Integer>    Width 80
}

Event: b.pressed()

After: Object {
 <String>     Name b
 <Integer>    X 180
 <Integer>    Width 80
}
```

```
Before: Object {
 <String>     Name b
 <Integer>    X 250
 <Integer>    Width 120
}

Event: b.pressed()

After: Object {
 <String>     Name b
 <Integer>    X 370
 <Integer>    Width 120
}
```

```
Event:
     b.pressed()

Source Variables:
     Integer b.X         [100 250]
     Integer b.Width     [80 120]

Target Variables:
     Integer b.X         [180 370]
```

```
Transition b.pressed()
{
     b.X := b.X + b.Width;
}
```

Figure 4-28: Source and Target Variables

Target variables are the variables that have to be solved. Attributes become target variables if they change from any Before snapshot to a corresponding After snapshot.

The source and target variables are the input to the “Inferencer” which tries to derive each target variable from a combination of source variables and constant values. If it succeeds, it produces a transition which contains an assignment for each target variable.

Inferencing - the search for relationships between variables - is inherently expensive. The Compactor eliminates irrelevant information so that the computationally much more expensive Inferencer is given the least possible information. In that way, inferencing is efficient even for user interfaces which contain many objects. In the above example, there could be many other interface elements besides the button for which functionality is demonstrated but the input to the inferencer would remain identical if the other elements are not touched during the demonstration.

There are two ways to find out which variables changed between snapshots. Ideally, the interface builder allows us to be notified of each individual change that the user makes when preparing the snapshots. In this case, we can directly find out which variables changed between snapshots, which would make the Compactor obsolete.[60]

Unfortunately, many interface builders do not allow external access to individual modification events. However, all of them are able to export information about the complete current state - the lowest common denominator is a file format for storing and retrieving designs. The Compactor can then efficiently recover what the user has changed between snapshots from these complete states.

---

60. Note that we prefer a snapshot-based approach over a trace-based approach for our purposes independently of what the underlying interface builder provides. This is mainly because a snapshot-based approach can better describe that changes to the user interface occur instantaneously upon events. It also avoids problems with users re-ordering and un-doing actions that are common to the trace-based approach.

Source variables are those that changed between any two Before snapshots. Source variables take their values from the Before snapshots. The Compactor identifies them by the following process.

It first constructs a vector of values for each attribute. This attribute becomes a source variable if the vector's minimum value is not approximately equal to the vector's maximum value. "Approximately equal" is defined for each type of attribute (we will discuss "inference types" in Section IV.4.1.2). For example, strings may be required to be exactly equal but screen coordinates are allowed to differ by up to fifteen pixels.

In Figure 4-28, the value vector of attribute *X* is [100 250] in the Before snapshots. It becomes a source variable because 100 is not approximately equal to 250. The attribute *Name* does not become a source variable because the elements of its value vector, [b b], are approximately equal.

Identifying source variables is linear in the number of attributes in the Before snapshots assuming that accessing attributes by name takes constant time (hash-based access).

No attribute can become a source variable if the demonstration consists of a single example because a single value is always approximately equal to itself. This is intended - there is no point in designating source variables because the Inferencer, described below, will always solve single-example demonstrations by assigning constant values to the target variables.

Target variables are those that change from any Before snapshot to a corresponding After snapshot. These variables take their values from the After snapshots. The Compactor constructs target variables in two phases - it first identifies target variables and then collects their values.

The Compactor identifies attributes as target variables by comparing the value of each attribute in a Before snapshot to its value in the corresponding After snapshot. The attribute is added to the target variable list if the two values are not (exactly) equal.

In Figure 4-28, the attribute *X* becomes a target variable because its value changes from a Before to a corresponding After snapshot. For example, it changes from 100 to 180 in the first example. Attribute *Width* does not become a target variable because it never changes its value in response to an event. It remains 80 in the first example and 120 in the second example.

The Inferencer is the component which relates target variables to source variables. It first groups the source and target variables by type. It then tests each target variable against unordered sets of source variables of the same type. A single such test checks if the target variable can be computed from a combination of the source variables. These tests are specific for each inference type. The Inferencer itself does not contain knowledge about types, it simply calls the test that is supplied by the inference type (Section IV.4.1.2).

The size of the sets increases over time. Testing ends if (1) a test succeeds, (2) the target variable has been tested against all unordered sets, or (3) a type-specific limit on set sizes is reached.

In the demonstration of Figure 4-28, the Inferencer first tests the target variable b.X against the empty source variable set. A test against an empty set succeeds if the target variable can be solved by a constant (e.g. b.X := 100), which is not the case here. The Inferencer then tests against the single-member sets {b.X} and {b.Width}. These tests also fail. The test against the set {b.X, b.Width} succeeds as shown in Figure 4-28.

The Inferencer's running time is worst-case exponential in the number of source variables of the same type. This does not present a problem because no variable becomes a source variable unless it is explicitly changed by the user during a demonstration. In our experience, demonstrations which would bring the Inferencer to its knees (more than, say, fifteen source variables of the same type) do not occur in practice because they would simply require too much effort to demonstrate in the first place. Differently stated, problems of such complexity are not suitable for a (domain-independent) demonstrational approach.

IV.4.1.2 Inference Types

The code which tests a target variable against a set of source variables is provided by the inference type. An inference type is an abstraction that contains all inferencing particular to an attribute type (such as “string” or “boolean”). That way, the “Inferencer” discussed above does not have to depend on specific types, and it is easier for a software engineer to later add additional types. Figure 4-29 shows the actual inference types we used in our prototype implementation.

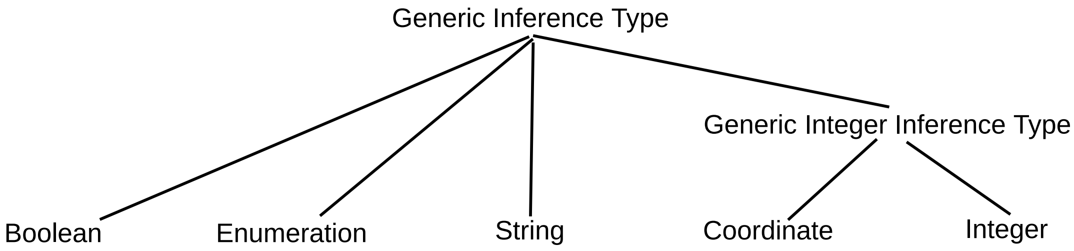


Figure 4-29: Inference Types Used in the Prototype Implementation

An inference type provides three methods (C++ functions in our implementation). The first method tells if two values of that type are “approximately equal”. The second method provides an “average value” given a list of values of that type. The third method tests if a given target variable can be solved by a combination of the source variables, and returns an assignment statement representing the solution if the test succeeds. Finally, each inference type contains a list of other types that it can implicitly be converted to (see Section IV.4.1.2.3).

In Figure 4-29, both the Coordinate and the Integer Inference Type inherit the “test” and “average value” method from an abstract superclass created for that purpose. However, they provide different “approximately equal” methods. The Integer inference type must be exactly equal, while Coordinates are allowed to differ by up to fifteen pixels to compensate for approximate placement during a demonstration.

The generic integer inference type contains by far the most complex reasoning for testing a target variable against a list of source variables, so that we will discuss it in more detail than the others.

IV.4.1.2.1 The Generic Integer Inference Type

The generic integer inference type can derive a target variable from a linear combination of source variables. That is, given target variable t and source variables s_1 to s_n it can determine the relationship $t = c_1s_1 + c_2s_2 + ... + c_ns_n + c_0$ given n+1 substantially different examples.[61]

There is a trade-off between the inferencing power and the responsiveness of any demonstrational system. We chose to restrict ourselves to linear relationships in the generic (domain-independent) reasoning because it covers a wide variety of user interface behavior (e.g. all of the behavior of the “MacDraw” imitation of Appendix D) while still allowing the system to respond immediately to demonstrations. It is entirely possible to extend the reasoning to cover additional relationships. However, doing so must be carefully balanced with the response time of the system. Super-linear inferencing may best be addressed by introducing domain knowledge into the reasoning process (see Section IV.5.3).

The algorithm that tests integer target variables against source variables works as follows. If the set of source variables is empty, the algorithm computes

---

[61] “Substantially different” is synonymous with “linearly independent” for the integer inference type.

the arithmetic mean of the values of the target variable and tests if this constant is a solution.

For example, if the target variable has the values [18 17 22] in a three-example demonstration, the algorithm computes the arithmetic mean, 19, and then tests if the vector [18 17 22] is approximately equal to [19 19 19]. If it is, the algorithm has solved the target variable (t:=19).

If the set of source variables is not empty, the algorithm constructs a matrix and a vector that can then be solved by Gaussian elimination. Assume there are *n* source variables $s_1 ... s_n$. If there are less than *n* examples, the test fails. If there are exactly *n* examples, the algorithm tries to derive the target variable from the source variables without an additive constant ($t = c_1 s_1 + c_2 s_2 + ... + c_n s_n$). If there are more than *n* examples it tries to solve the general case ($t = c_1 s_1 + c_2 s_2 + ... + c_n s_n + c_0$).

This is again best explained through examples. Consider the introductory example of Figure 4-28, where the variables are as follows.

```
Source Variables:
     Integer b.X [100 250]
     Integer b.Width [80 120]

Target Variables:
     Integer b.X [180 370]
```

The algorithm takes these examples and constructs the following matrix. The columns of the matrix correspond to the values of the source variables (b.X and b.Width here), respectively. The vector is made up from the values of the target variable that we are trying to solve (b.X here).

$$\begin{bmatrix} 100 & 80 \\ 250 & 120 \end{bmatrix} \times \begin{bmatrix} \text{b.x} \\ \text{b.width} \end{bmatrix} = \begin{bmatrix} 180 \\ 370 \end{bmatrix}$$

Standard Gaussian elimination can then be used to solve this set of equations.

$$\begin{bmatrix} \text{b.x} \\ \text{b.width} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

That is, we have found a relation between the target variable and the source variables, namely

$$\text{b.X} \leftarrow 1 \cdot \text{b.X} + 1 \cdot \text{b.Width}$$

The inference engine then checks if this is indeed a solution by re-substitution[62]. A simplifier brings the solution to its final form, b.X ← b.X + b.Width.

All examples in this paper use demonstrations which can be solved exactly for the sake of simplicity. We discuss later how we use snapping to deal with approximate solutions such as “b.x =1.03, b.width = 0.98”

In the previous example, there were no redundant examples (“bad examples”), and there were only *n* examples for *n* variables. The following example shows how the algorithm constructs the matrix in a more general case.

---

[62] Re-substitution would not be necessary if the exact solution is used, of course. However, we also use re-substitution to test if a “snapped” version of the solution will do as explained later.

In this example, assume that the user has given several demonstrations that center interface object *b* between objects *a* and *c* by moving *b* (rather than resizing *b*). Figure 4-30 shows the variables relevant to the demonstration.

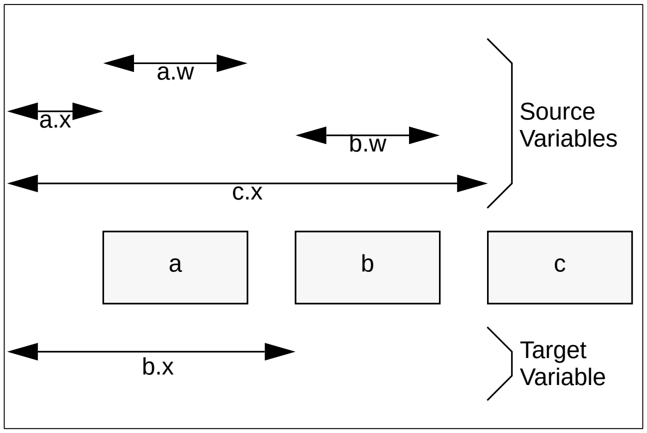


Figure 4-30: Centering Elements

The formula for *b.x* which centers *b* in this way is shown below.

$$b.x \leftarrow a.x + a.w + \frac{1}{2}(c.x - (a.x + a.w)) - \frac{1}{2}b.w$$

Let us assume that the demonstration consists of seven examples as shown in Table 4-1.

Table 4-1: Example Values for Integer Inferencing

Example:	1.	2.	3.	4.	5.	6.	7.
source variable a.w	200	200	100	100	50	100	200
source variable a.x	200	200	200	0	0	100	200
source variable b.w	100	100	200	200	50	50	100
source variable c.x	800	600	800	800	150	300	800
target variable b.x	550	450	450	350	75	225	550

If we have more examples than needed, which ones should we select? Ideally, we want to use the most “unique” examples. Seen from a demonstrational standpoint, one intuitively feels that using nearly identical examples will not give the inference engine more useful information. Seen from a “math” standpoint, one does not want to have rows in the matrix that are nearly identical because that increases the likelihood that one row is linearly dependent on the others (that the augmented matrix is unsolvable).

We use the following method to select examples that will be used in the matrix[63]. We define the *distance* between two examples as the count of source variables that are not approximately equal between them. We define the *uniqueness* of an example as the sum of its distances to the other examples. We then select *k* examples of decreasing uniqueness for inclusion in the matrix (where *k* is the number of source variables plus one). Table 4-2 lists the distances for each example in Table 4-1.

For example, the distance between the first and second example in Table 4-1 is one because *c.x* is the only source variable with a differing value. The

---

[63] There is actually a superior selection method for the integer inference type. This method consists of adding vectors to a new matrix one by one, making sure that every new vector is linearly independent from the ones already there. We present the above method because it can be used for all inference types, not just for integers.

Table 4-2: Computing the "Uniqueness" of Examples

Example:	1.	2.	3.	4.	5.	6.	7.
1.	-	1	2	3	4	4	0
2.	1	-	3	4	4	4	1
3.	2	3	-	1	4	3	2
4.	3	4	1	-	3	3	3
5.	4	4	4	3	-	4	4
6.	4	4	3	3	4	-	4
7.	0	1	2	3	4	4	-
Uniqueness:	14	17	15	17	23	22	14

distance between the first and fifth example is four because all four source variables differ between them.

Thus, we select examples two through six to construct the matrix below. In this demonstration, we have more examples than source variables which allows us to add the row of ones which tests for a constant offset (e.g. p.x := q.x + 100).

$$\begin{bmatrix} 200 & 200 & 100 & 600 & 1 \\ 100 & 200 & 200 & 800 & 1 \\ 100 & 0 & 200 & 800 & 1 \\ 50 & 0 & 50 & 150 & 1 \\ 100 & 100 & 50 & 300 & 1 \end{bmatrix} \times \begin{bmatrix} a.w \\ a.x \\ b.w \\ c.x \\ const \end{bmatrix} = \begin{bmatrix} 450 \\ 450 \\ 350 \\ 75 \\ 225 \end{bmatrix}$$

Solving the matrix returns

$$\begin{bmatrix} a.w \\ a.x \\ b.w \\ c.x \\ const \end{bmatrix} = \begin{bmatrix} 0.50 \\ 0.50 \\ -0.50 \\ 0.50 \\ 0 \end{bmatrix}$$

which results in the following derivation after running it through the simplifier. The formula indeed centers *b* such that it has equidistant edges to *a* and *c*.

$$\mathrm{b.x} \leftarrow \frac{1}{2}\mathrm{a.w} + \frac{1}{2}\mathrm{a.x} - \frac{1}{2}\mathrm{b.w} + \frac{1}{2}\mathrm{c.x}$$

The numerical examples above all use equations that can be solved exactly. This will rarely be the case in actual demonstrations where raw solutions often read “b.x := 1.03*b.x + 0.48*b.width + 3.1” when the user intended “b.x := b.x + 1/2*b.width”.

We deal with these cases by first trying if the non-constant factors can be snapped to halves and the constant factor snapped to zero (which is the case in the example above).[64] We then try snapping the non-constant factors to halves and rounding the constant factor. If both these snapped versions of the solution fail we use the original solution.

$$\begin{array}{ccc} \text{First Try} & \text{Second Try} & \text{Third Try} \\ \begin{bmatrix}1.03\\0.48\\3.1\end{bmatrix} \rightarrow \begin{bmatrix}1\\0.5\\0\end{bmatrix} & \begin{bmatrix}1.03\\0.48\\3.1\end{bmatrix} \rightarrow \begin{bmatrix}1\\0.5\\3\end{bmatrix} & \begin{bmatrix}1.03\\0.48\\3.1\end{bmatrix} \end{array}$$

---

[64] Such a trial consists of resubstituting the snapped solution into the vector on the left-hand side and checking if this results in a solution vector that is “approximately equal” to the desired solution vector (“approximately equal” is defined for each inference type as explained in Section IV.4.1.2).

IV.4.1.2.2 Other Inference Types

As shown in Figure 4-29, we defined three more inference types beyond the two integer-based inference types, namely Boolean, Enumeration and String. Not much is to be said about the Boolean and Enumeration inference types as they simply subclass from the Generic Inference Type without providing additional reasoning capabilities.[65] That is, the inferencing mechanism can still only infer assignment from a constant value or assignment from another variable of the same type. The String inference type additionally tests if a target variable can be solved by concatenating two String source variables.

The decision to only provide very modest reasoning capabilities for types other than integers reflected the requirements for our prototype implementation. We see no inherent difficulty in expanding the reasoning capabilities of the existing inference types, or in adding new inference types.

IV.4.1.2.3 Type Conversion

As mentioned at the beginning of Section IV.4.1.2, inference types also contain a list of other types that they can be converted to. For example, we found it useful to define Strings as being implicitly convertible to Integers. For example,

---

[65] The sole reason for their existence is that printing out source or target variables of these inference types at debugging time will then better reflect their original types (the types of the attributes or event parameters from which they were constructed).

the line width of all currently selected objects can then be set to an integer value specified in a text field. (The line width of the objects are integer-valued target variables while the content of a text field is a string-valued source variable.)

Declaring such a type conversion has the simple effect that an additional artificial source variable is generated before the source variables are sorted by type. In the context of the example above, if the value of the text field *tf* was “3” in a first example and “7” in a second example, an artificial Integer source variable is generated as shown below.

```
Source Variables:
     String    tf.text   ["3" "7"]
     Integer   tf.text   [3 7]
```

The reasoning process otherwise proceeds as usual (Section IV.4.1.1).

IV.4.1.3 Inferring Set Expressions

We have so far talked about the mechanism that infers assignment statements, or more precisely the *value expressions* on the right-hand side of assignment statements, such as “*x + 1 / 2 * width*” (see Section III.2.3). This section explains the basic inferencing mechanism for *set expressions*, such as “(*bb.mode ==”RectangleMode” && *.class=="Rectangle")*” (see Section III.3.1). We will call this inferencing mechanism the “set expression finder”.

This mechanism is best introduced through an example. Suppose the user interface design consists of three elements called Doc1, Doc2 and Doc3, and further suppose that the designer has given two examples as follows. In the first example, clicking on “Doc1” deletes this element, and in the second example clicking on “Doc2” deletes that element.

Part of solving this demonstration is to find a set expression that describes which elements get deleted. Assume that the following set source variables are passed to the set expression finder by the sub-component for reasoning about deletions. (There is no need to understand what these variables “mean” for the purpose of explaining the set expression finder.[66])

```
<string> "*.name"    {Doc1 Doc2}  {Doc2 Doc3 Doc1 Doc3}
<string> "self.name" {Doc1 Doc2}  {Doc1 Doc1 Doc2 Doc2}
```

A single set source variable consist of an inference type name (“<string>”), the name an event parameter, concrete attribute, or wildcard attribute (“*.name”), a list of its values for the positive examples (“Doc1 Doc2”), and of a list of its values for the negative examples (“Doc2 Doc3 Doc1 Doc3”).[67] There are two set source variables in this example input.

---

66. The set expression finder is used by the reasoning for assignment statements, create statements, delete statements, and for the expression in the header of a transition. Each of these reasoning components is in charge of preparing the set source variables fed to the set expression finder. These components are discussed in Sections IV.4.2.1, IV.4.2.2, IV.4.2.3 and IV.4.3, respectively.

67. Note that the mechanism internally does not carry a list of actual values but rather pointers to contexts that allow to compute these values.

When the set expression finder is given this list of set source variables, it first generates a list of set expressions consisting of simple comparisons. In our concrete example, the following comparisons are generated.

```
(*.name==”Doc1”)          {1}       {-1 -2 -3}
(*.name==”Doc2”)          {2}       {-1 -3 -4}
(*.name==”Doc3”)          {}        {-2 -4}
(*.name==self.name)       {1 2}     {-1 -2 -3 -4}
(self.name==”Doc1”)       {1}       {-3 -4}
(self.name==”Doc2”)       {2}       {-1 -2}
```

In this notation, the generated single-comparison expression is shown on the left. The lists in curly parentheses indicate which of the six cases would be satisfied by that expression. (In this example, there are two positive and four negative cases.)

The next step is to remove set expressions that are dominated by others. A set expression is dominated if another expression solves a superset of its cases. In this particular example, the fourth set expression dominates all others.

Inferencing stops when we found an expression which solves all cases. In the above example, the fourth expression does, and we are done. The component that reasons about delete statements will now produce a delete statement which correctly deleted the intended element for the given examples.

```
Transition ...
{
     delete (*.name==self.name);
}
```

In this particular example, this expression is actually equivalent to a reference to the element on which the event occurred, so that the generated output is actually as follows, after being processing by an algebraic simplifier.

```
Transition ...
{
     delete self;
}
```

We will now present an example of a more complicated problem for the set expression finder. The user has demonstrated that dragging elements from the canvas to the trashcan element deletes them, while dragging them to any other element does not. Elements on the canvas are marked with a user-defined “onCanvas” attribute. The user provided two positive and three negative examples. The corresponding input to the set expression finder is shown below. (Note that this is a real-life example, where we have demonstrated the behavior to Grizzly Bear and traced the internal reasoning. There is no need to understand what these variables “mean” for the purpose of explaining the set expression finder.[66])

```
<string>  *.onCanvas  {yes yes}      {yes no no}
<string>  *.imageName {D.ic F.ic}    {D.ic D.ic D.ic}
<string>  *.name      {auto2 auto1}  {auto2 Doc Doc}
<coord>   *.x         {116 81        {116 96 96}
<coord>   *.y         {62 92}        {62 77 77}
<coord>   Doc.x       {45 45}        {45 96 96}
<coord>   Doc.y       {3 3}          {3 77 77}
<string>  over        {Trash Trash}  {Doc Trash Canvas}
<coord>   x           {217 220}      {63 218 154}
<coord>   y           {123 126}      {19 120 155}
```

As in the previous example, the set expression finder first generates a list of single-comparison expressions.

```
(*.onCanvas==”yes”)        {1 2}        {-2 -3}
(*.onCanvas==”no”)         {}           {-1}
(*.imageName==”D.ic”)      {1}          {}
(*.imageName==”F.ic”)      {2}          {-1 -2 -3}
(*.name==”auto1”)          {2}          {-1 -2 -3}
(!(*.name==”Doc”))         {1 2}        {-2 -3}
(*.x<88)                   {2}          {-1 -2 -3}
(!(*.x<106))               {1}          {-2 -3}
(*.y<69)                   {1}          {-2 -3}
(!(*.y<84))                {2}          {-1 -2 -3}
(Doc.x<70)                 {1 2}        {-2 -3}
(!(Doc.x<70))              {}           {-1}
(Doc.y<40)                 {1 2}        {-2 -3}
(!(Doc.y<40))              {}           {-1}
(over==”Trash”)            {1 2}        {-1 -3}
(over==”Doc”)              {}           {-2 -3}
(over==”Canvas”)           {}           {-1 -2}
(x<218)                    {1}          {-2}
(!(x<185))                 {1 2}        {-1 -3}
(!(x<219))                 {2}          {-1 -2 -3}
(y<69)                     {}           {-2 -3}
(y<140)                    {1 2}        {-3}
(!(y<121))                 {1 2}        {-1 -2}
```

The generated expressions are dependent on the inference type (there is a fourth method for inference types called “generateSeeds” which was not mentioned in Section IV.4.1.2). We generate “smaller-than” comparisons for the Coordinate inference type, “equal-to” comparisons for Enumerations, Booleans and Strings, and both kinds of comparisons for Integers.

The negated clauses of generated set expressions are also added if they are not already dominated by existing expressions. After eliminating expressions that are dominated by others, we are left with the following candidates.

```
(*.onCanvas==”yes”)        {1 2}        {-2 -3}
(*.imageName==”F.ic”)      {2}          {-1 -2 -3}
(over==”Trash”)            {1 2}        {-1 -3}
(!(y<121))                 {1 2}        {-1 -2}
```

Because none of these expressions presents a solution (that is, none solves both positive and all three negative cases) we go through the following expansion process. We add two-comparison terms by “oring” and “anding” existing expressions together if the resulting expression solves a set of cases not already solved by an expression in the list. We also again add their negated versions if they pass the same test. This leaves us with the following candidates in this particular example.

```
(*.onCanvas==”yes”)                         {1 2}   {-2 -3}
(*.imageName==”F.ic”)                       {2}     {-1 -2 -3}
(over==”Trash”)                             {1 2}   {-1 -3}
(!(y<121))                                  {1 2}   {-1 -2}
((*.onCanvas==”yes”)&&(over==”Trash”))      {1 2}   {-1 -2 -3}
((*.onCanvas==”yes”)&&(!(y<121)))           {1 2}   {-1 -2 -3}
((over==”Trash”)&&(!(y<121)))               {1 2}   {-1 -2 -3}
```

The reduce-check-expand cycle now begins again. We are left with the expression below after the reduction step, which presents a solution in this particular example. (This expression was part of a demonstration that expressed that objects on the canvas can be deleted when they are released over the trashcan.)

```
((*.onCanvas==”yes”)&&(over==”Trash”))      {1 2}   {-1 -2 -3}
```

The listing of candidate expressions is always ordered in terms of desirability - we prefer simpler solutions to more complex solutions. Being “simpler” is defined in the number of nodes in an expression tree. The seven expressions above have a node size of 3, 3, 3, 4, 7, 8, and 8, respectively.

If expressions have the same “simplicity”, we pick the one which happens to appear first in the expression list. The one-comparison “seed expressions” are originally ordered alphabetically, so that there is a bias towards variables that rank lower in the alphabet.

Finally, let us note that this algorithm is closely related to work in machine learning, more specifically to work on “learning by similarity detection” (see e.g. [Kodr88], chapter 8). Some of these algorithms exhibit a better running time for certain special cases of the problem of finding set expressions. (Our algorithm is exponential in the number of set source variables in the worst case. This has not been a problem for us because our source variable technique ensures that there are never large input sizes.) There are also some characteristics of our domain (most notably wildcard slot identifiers) which have kept us from reusing an existing machine learning algorithm. An algorithm by Michalski (also in [Kodr88], chapter 8) seems to be the one most closely related to ours. We may try to build on it in the future if the performance of the set expression finder becomes an issue.

IV.4.2 Inference Bear’s Inferencing

Inference Bear’s internal reasoning consists of three major sub-components, one each for the modification, creation, and deletion of elements. Inference Bear first computes the source variables for the demonstration (to avoid wasteful re-computation in the sub-components). It then calls each of the three components, and concatenates the statements returned by the components to make up the transition body (the statements concerning the creation of elements come first, then the ones for modification, then the ones for deletion). Finally, it synthesizes a transition header that makes the behavior dependent on the literal event used in the examples.

Section IV.4.2.1 now presents the reasoning for the modification of elements in more detail. Section IV.4.2.2 is concerned with the creation of elements, and Section IV.4.2.3 is concerned with the deletion of elements

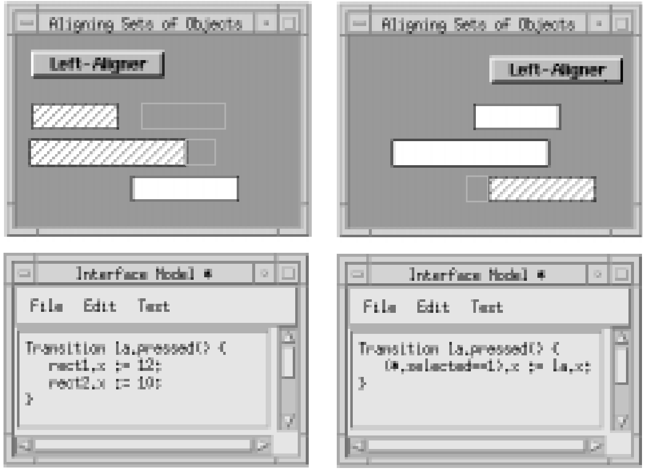
IV.4.2.1 Changing Attributes of Existing Elements

The sub-component of Inference Bear concerned with demonstrations that modify attribute values first tries to find a “conventional” solution for each target variable separately as already discussed in Section IV.4.1.

If a collection of target variables cannot be solved “conventionally”, that is by considering each variable separately, we use the following mechanism to detect if there is a “set solution”, namely an assignment statement which contains a set expression on its left-hand side rather than an absolute reference to an element (see Section III.3.1).

We first group the target variables by attribute name, and then consider each group separately. In each group, we use the set expression finder (Section IV.4.1.3) for inferring the set expression that goes into the left-hand side of an assignment, and re-use the single-attribute reasoning (Section IV.4.1.1) for the value expression that goes into the right-hand side of the assignment.

That process is again best explained by an example. Assume 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. (Selected objects are highlighted by diagonal stripes. The dotted white shadows in Figure 4-31 indicate where the moved objects were located in the *before* snapshot.)



```
Transition la.pressed() {
   rect1.x := 12;
   rect2.x := 10;
}
```

First Example.

```
Transition la.pressed() {
   (*.selected==1).x := la.x;
}
```

Second Example.

Figure 4-31: Aligning Sets of Objects

We first mark the examples in which the target variables did not change from a *before* to an *after* snapshot with a “don’t care” symbol (a hyphen). The (slightly idealized) input to the set reasoner for this example is as follows.

```
Boolean Source  Variable      rect1.selected     [1 0]
Boolean Source  Variable      rect2.selected     [1 0]
Boolean Source  Variable      rect3.selected     [0 1]
```

```
Coord   Source  Variable      la.x                [10 90]
Coord   Target  Variable      rect1.x             [10 -]
Coord   Target  Variable      rect2.x             [10 -]
Coord   Target  Variable      rect3.x             [- 90]
```

The next step is to construct an artificial set source variable that is made up from the source variables available for our target variable group.[68] We then call the set expression finder for that artificial set source variable. The presence of a value in the target variable at that position presents a positive case and a “don’t care” value presents a negative case. The input to the set expression finder is shown below.

```
<boolean> *.selected {1 1 1} {0 0 0}
```

The set expression finder then derives the expression “(*.selected==1)” from this input as explained in Section IV.4.1.3, and we are done as far as the left side of the assignment is concerned.

We now generate an artificial target variable by taking the values from individual variables for each example and by synthesizing an “average value” for them if they are “approximately equal”.[69] In our example, this results in the following target variable.

---

[68] Note that if e.g. only “rect1.selected” would be in the original source variable set, we would automatically add “rect2.selected” and “rect3.selected” here as additional source variables.

[69] These two functions are defined for every inference type, see Section IV.4.1.2.

```
Coord   Target  Variable      artificial.x       [10 90]
```

We then use our standard mechanism to test this target variable against the original source variables, and additionally against similarly manufactured “self” source variables. If that succeeds, we are done, and can now synthesize the final assignment statement, as shown below.

```
(*.selected==1).x := la.x;
```

The demonstrations presented in Sections IV.1.4, IV.1.5, and IV.1.6 involved this mechanism. The mechanism can also handle references to “self” in the right-hand side of an assignment, in the manner briefly mentioned above. For example, the designer can demonstrate that clicking on any circle will put all rectangles 50 pixels below the circle that was clicked:

```
Transition (*.class=="Circle").pressed()
{
        (*.class=="Rectangle").y := self.y + 50;
}
```

IV.4.2.2 Creating Elements

Inference Bear’s sub-component for reasoning about newly created objects first makes a list of elements that have been created by comparing each *after* snapshot to the corresponding *before* snapshot. It then guesses which elements of e.g. the first example correspond to the elements in the second example by clustering the elements with the fewest differing attributes. (Imagine that you created a rectangle and a circle in both examples. Because of our snapshot-based technique, the order in which they appear in the list of newly created elements may differ between examples, this is why the clustering is needed)

For each new element, we choose an existing element of the same class as the prototype for the new element, namely the one with the fewest differing attributes. Any attribute of the new element that changed from that prototype is treated as a target variable, and an assignment for it is generated using the standard mechanism (Section IV.4.2.1).

In the above algorithm, all prototypes are explicitly named elements. If this algorithm fails (for example, if we want to describe that “clicking on a canvas produces a copy of the currently-white element”), we find the prototype with the fewest differing attributes for each example separately, and then construct a set expression for the create statement by treating the prototype elements as positive cases, treating all other elements as negative cases, and calling the set expression finder (see Section IV.4.1.3).

This algorithm cannot currently handle the creation of a dynamic number of elements. Indeed, this cannot even be expressed in the Elements, Events & Transitions language itself because a single create statement always creates exactly one element (this situation has to be handled via the Application Programmer Interface at the moment). We could not identify a (domain-independent) algorithm that can handle this inference efficiently; however, such an extension is clearly desirable to handle common user interface behavior (“copy all currently-selected elements and place the new copies at an offset from the original ones”). This may present a case in which it is desirable to add domain-specific code to augment our inferencing.

IV.4.2.3 Deleting Elements

Inference Bear’s sub-component for the deletion of elements first constructs a list of pointers to all elements that have been deleted as well as another list of pointers to all elements that have not been deleted (by comparing the *before* and *after* snapshots). It then passes the problem on to the set expression finder (Section IV.4.1.3) by constructing set source variables which treat the deleted elements as positive cases and the other elements as negative cases. The resulting set expression is then put into the delete statement (e.g. “delete (*.selected==1)”).

For example, consider the scenario of the first example in Section IV.4.1.3, in which a simplistic user interface contains three elements called Doc1, Doc2 and Doc3. The user has given two (positive) examples. In the first example, clicking on Doc1 deletes this element while in the second example clicking on Doc2 deletes that element. Part of solving this demonstration is to find an expression for the elements which get deleted. This is done by computing set source variables which treat the deleted elements as positive examples and all others as negative examples.

In this example, Doc1 was deleted in the first example but not Doc2 or Doc3, while Doc2 was deleted in the second example but not Doc1 or Doc3 (the values stemming from the first example are shown in bold, the values stemming from the second example in italics).

`<string> "*.name"` {**Doc1** *Doc2*} {**Doc2 Doc3** *Doc1 Doc3*}

The deletion sub-component then computes the source variables from the before snapshots as usual (Section IV.4.1.1), and converts them to set source variables. This is done by multiplying their value for each example by the overall number of elements in the example. For example, the name of the object on which the event occurred changed between the first and second example (Doc1 versus Doc2), so that it becomes a source variable. It is then converted to a set source variable as follows.

`<string> "self.name"` {**Doc1** *Doc2*} {**Doc1 Doc1** *Doc2 Doc2*}

The set source variables are then passed to the set expression finder, which computes "*delete (*.name==self.name)*" as a solution for this particular example (which is equivalent to "*delete self*", see Section IV.4.1.3).

Note that treating *all* other elements as negative cases is both conservative and expensive - it makes sure that the inferred set expression does not accidentally match an unrelated element at the expense of inferencing speed. We did not experience a performance problem with this approach for our limited designs (less than one hundred elements overall), but this is easily imaginable for user interfaces consisting of thousands of elements. In such a case, it is possible to use a heuristic for preventing unrelated elements to accidentally match the expression in a delete statement, such as making the deletion of an element dependent on being on the same canvas as the elements in the original demonstration (by attaching an "*and* it is on canvas *x*" clause to the inferred expression at demonstration time). This way, there is no need to treat all elements as negative cases during a demonstration but rather only the ones on the same canvas as the deleted elements.

IV.4.3 Expression Finder's Inferencing

As long as there are no negative examples the Expression Finder simply concatenates the names of the elements on which the events occurred, using logical "or" (e.g. "(*.name=='circle' || *.name=='ellipse').pressed()").

Otherwise, the Expression Finder is a one-to-one front-end to the set expression finder described in Section IV.4.1.3. It constructs the set source variables using each positive example as "positive case", and each negative example as a "negative case" (in the terminology of the set expression finder). The output from the set expression finder then becomes the header of the generated transition.

IV.4.4 Grizzly Bear's Inferencing

There is no additionally reasoning that is particular to Grizzly Bear. It simply first internally calls Inference Bear, feeding it only the positive examples, then calls the Expression Finder, feeding it everything but the *after* snapshots, and finally combines the transition body from Inference Bear's output with the transition header of Expression Finder's output.

IV.5 Discussion of the Demonstrational Components

We will discuss the limitations of our demonstrational tools in conclusion of this chapter, and also compare the relative advantages and disadvantages of our "domain-independent" approach. Finally, we will speculate on how to introduce domain knowledge in a principled way.

IV.5.1 Limitations

In this section, we will characterize behavior that cannot be inferred by Grizzly Bear but is nevertheless relevant for building user interfaces.

First and foremost, Grizzly Bear does not deal with text other than simple labels, so that the behavior of textual windows has to be defined by other means. Grizzly Bear also cannot integrate pre-built complex components (such as a text editing widget or a file selection box) with the rest of a design.

There are also limitations to the complexity of transitions it can infer. For example, it cannot recognize the creation of a dynamic number of elements (see Section IV.4.2.2). It is also not well-suited for user interfaces in which many elements continuously stay attached to each other, primarily because its underlying language currently lacks constraints [Myer90b,Huds93].

IV.5.2 Domain-Specific PBD Systems vs. Domain-Independent PBD Systems

There are two competing approaches towards building Programming By Demonstration (PBD) systems, which we will call the “domain-specific” and the “domain-independent” approach for lack of a better name. We have at times also referred to domain-specific systems as “rule-based systems” or “expert systems” while we called their domain-independent counterparts “algorithm-based” or “more mathematically-thorough”.

The distinction between those two is not clear-cut. Systems that we call domain-specific typically check a given demonstration against a finite list of situations they recognize, while domain-independent systems rely on machine learning techniques for their inferencing.

While the following definition does not cover all systems that we would intuitively call domain-specific, we define such PBD systems as *systems that refer to particular attributes by name in their reasoning*. That is, the attributes the system can reason about are determined when the system is built. We will call all other systems “domain-independent”.

For example, a domain-specific system may contain a rule that specifically checks if a new object is centered relative to an existing one by checking their attributes named, say, “x-pos”, “y-pos”, “width” and “height”. Similarly, another rule can check if a new object is left-aligned by checking if the values of their attributes named “x-pos” are similar. For example, Peridot [Myer88] and Druid [Sing90] have taken this approach.

A possible domain-independent approach is to instead check if the attributes of the new object can be computed from a linear combination of existing variables. This was our approach. Other systems of a similar nature are Demo [Wolb91] and GITS [Olse90].

We will discuss the advantages and disadvantages of the two approaches using the two main characteristics of PBD systems: how much they can infer, and how easy they are to use.

IV.5.2.1 Inferencing Power

From the above example, it may seem that domain-independent systems would always be able to draw the same inferences that domain-specific systems can draw, and more. This is not the case. Domain-independent systems are superior from an inferencing standpoint only to domain-specific systems whose rules are simply special cases of variables that are linearly dependent on existing variables.

Domain-specific systems can check for relationships that are not linear. For example, they can have a built-in rule that checks for a parabolic trajectory. The reason that domain-independent systems cannot discover general super-linear relationships is because this is computationally too expensive. Domain-specific systems do not attempt to solve the general problem, but rather just solve super-linear demonstrations of particular interest by codifying prior knowledge of what types of relationships are going to be demonstrated.

An important inferencing advantage of domain-independent systems is their ability to deal with user-defined variables. For example, our Grizzly Bear (Section IV.3) can reason about attributes added interactively by the user just as it would reason about any other attribute. Domain-specific systems cannot make this inference as they - by our definition - base their reasoning on knowledge about the roles of particular attributes that they identify by name (and the names of user-defined attributes cannot be known in advance).

IV.5.2.2 Ease of Use

The theoretical inferencing power of a PBD system is of no relevance if its potential users are not able to provide it with correct demonstrations. We believe that neither approach is inherently “easier to use” than the other.

We believe that domain-independent systems are easier to use in the sense that they are more predictable. In our particular system, demonstrating that a variable takes its value from another variable takes the same type of demonstration, no matter if the variables are colors, positions, labels, fonts, or so on. Because of the rule-based reasoning of domain-specific systems, it is easily possible that a seemingly similar demonstration results in a different type of inference because a different rule fired (or none at all). The user is then left to wonder if a slightly different demonstration would have brought the desired result, or if the system simply has no rule built-in for this situation.[70]

On the other hand, domain-specific systems often require only a single example because they, in some sense, already know what is going to be demonstrated. Informally stated, they can thus jump in after the first example and ask “I know this! You are trying to do *x*, right?”. Having to give only a single example is a significant usability advantage as it is our experience that going from single-example demonstrations to multiple-example demonstrations is difficult for new users.

Another advantage of domain-specific systems is that their rule-based structure invites refinement based on user studies. Subjects can first be observed to find out how they *think* a particular behavior should be demonstrated (which is all that matters).[71] The rules can then easily be changed to indeed translate these demonstrations into the expected behavior.

IV.5.2.3 Conclusion

In conclusion of the “domain-independence” discussion, we believe that a domain-specific approach is the right choice for PBD systems that are concerned with small-scale tasks in which the behavior to be demonstrated is predictable and can be covered by a small set of rules. We see their main advantage in the ease with which they can be tuned to their users’ expectations.

---

[70] Note that domain-independent systems can of course also fail to produce the desired result - it is just that their behavior is more predictable.

[71] Assuming, of course, that all users think along the same lines - which may or may not be the case.

We advocate the domain-independent approach for open-ended tasks such as designing functional graphical user interfaces because of its breadth of coverage. We also believe that a domain-independent approach better facilitates the demonstration of innovative behavior because it does not have to codify assumptions about what is going to be demonstrated. Finally, we believe that domain-independent PBD approaches present a more exciting research direction because they potentially enable the transfer of inferencing technology from one PBD system to another.

IV.5.3 Integrating Domain Knowledge

The best strategy for future PBD systems may be to combine both approaches by first checking if domain-specific rules fire and by then falling back on a domain-independent reasoning engine such as ours. At least one existing demonstrational system has used such a hybrid approach [Fish92].

Our inferencing engine also already provides a number of “hooks” (C++ function calls) which let one tune the inferencing based on domain knowledge. For example, one such hook provides for sorting source variables by domain-specific criteria. We have used this hook in our prototype implementation to put the “x” and “y” event parameters that correspond to the location of an event at the end of that list. This is because it is nearly always possible to tell positive examples from negative examples by the precise location of the event in the positive examples -

putting them at the end of the list ensures that the inferencing mechanism only uses this solution as a last resort.

CHAPTER V

BUILDING MODELS BY INTERVIEW

The original plans for our user interface design environment called for less sophisticated demonstrational tools in favor of an additional component called the “interview tool”. We envisioned that this tool would help novice designers construct and complete user interface designs by asking them a series of questions, and by building up a textual model of the user interface based on their answers.

During the course of the research, the inherent limitations of this approach became apparent, so that we no longer advocate a rule-based, heuristic interview component as a general methodology for building user interfaces. The main problem was that our domain (two-dimensional graphical user interfaces) turned out to be too large and open-ended to accommodate capturing generally-valid design advice in an expert-system fashion.

V.1 The Role of an Interview Component

We now see the role of an interview component in user interface design as more limited, similar to the role of Microsoft’s “wizard” components. These “wizards” are similar to our “interview tool” in intent and approach.[72] For example, the “layout wizard” in Microsoft Word asks the user a series of questions and then constructs an initial design based on the answers. Just as in the interview tool, the answers are given in the form of selections from multiple choices, not in the form of unrestricted natural language.

An interview component can be genuinely useful in assisting with easily-anticipated tasks (for example, we can anticipate that office workers will often want to set up a mail-merge between Microsoft Excel and Microsoft Word). It is just that an interview approach can never present a *general* design methodology for any non-trivial application domain.[73]

The remainder of this chapter contains the original design of the interview component as presented in [Fran93]. However, the text has been updated to reflect our current views.

The underlying modelling language is a precursor of the Elements, Events & Transitions (EET) model that was inspired by UIDE [Fole89]. From a computational standpoint, the EET model is a superset of that earlier version: “actions” were replaced by “transitions”, pre-conditions were subsumed by transition headers, and post-conditions were subsumed by transition bodies.

---

72. The systems were developed independently. One difference is in the internal structure of the tools’ knowledge: the “wizards” use arbitrary programming language code while the “interview tool” uses rules (which facilitates the maintenance of the knowledge base).

The paper refers to the user interface design environment as “Interactive UIDE”, as its motivation was to augment previous research on UIDE with interactive design tools.

V.2 The Design of the Interview Tool

We will first briefly introduce the underlying modelling language by presenting a small partial model of a chess application. Keywords are shown in bold face.

```
Boolean gameExists := false
Boolean unsavedChanges := false

Action New
         Precondition  “!unsavedChanges”
         Postcondition “gameExists := true”
Action Open
         Precondition  “!unsavedChanges”
         Postcondition “gameExists := true”
Action Discard
         Postcondition “gameExists := false;
                        unsavedChanges := false”
```

---

73. Imagine that you could change fonts *exclusively* through a “wizard” in Microsoft Word. If you then had to change some text to a specific font such as “10-point times roman” (say because this font is prescribed by a publisher) you would have to try and get the “wizard” to choose this font using a trial-and-error strategy. That is, you would have to try if some combination of answers to its questions makes it choose that specific font for the main body of the text (which may or may not be the case). Such forced usage of an interview component is obviously undesirable.

```
Action Save
         Precondition  “gameExists and unsavedChanges”
         Postcondition “unsavedChanges := false”

Action Quit
         Precondition   “!unsavedChanges”

Class ChessPosition
         Enumeration horizontal: ‘a’..’h’;
         Enumeration vertical: 1..8;

Class ChessPiece
         Variable ChessPosition pos;
         Action move(ChessPosition p)
               Postcondition “pos := p;
                              unsavedChanges := true”

Class King SubclassOf ChessPiece
Class Queen SubclassOf ChessPiece
Class Rook SubclassOf ChessPiece
Class Bishop SubclassOf ChessPiece
Class Knight SubclassOf ChessPiece
Class Pawn SubclassOf ChessPiece
```

The first two elements defined in this model are variables which maintain a subset of the application state, namely if a chess game exists and if it contains unsaved changes. The next elements describe user actions which are accessible from the user interface. Actions are only available if their preconditions apply. In our example, the “New” action is only available if there are no unsaved changes to the current game. The postconditions are assertions which modify the state of the application. For example, the “Discard” action resets the chess application to its initial state. Classes group semantically related data and actions relevant to the user interface. In this model, a ChessPiece is an entity which has a position on the board (data) and which can be moved to a new position (action). Classes can be organized hierarchically. In our example, the King is a particular kind of a Chess-Piece.

The intent of our application modelling language is to capture the application elements which are relevant to the user interface. It does not provide for multiple inheritance, virtual functions or different inheritance flavors. Our objective was not to develop a more complete object-oriented structuring language but to rather have a sensible compromise between expressiveness and novice understandability.

The declarative sequencing through pre- and postconditions facilitates reasoning by external tools. These tools can “understand” the sequencing to an extent that would not be possible with a general-purpose programming language. For example, a help generator can do backchaining to find a sequence of actions which enables an unavailable action by recursively evaluating pre- and postconditions. This sequence can then be presented in textual or animated form.

V.2.1 Modes Of Interactive UIDE

At design time, the designer can concurrently edit the user interface, the application model and the “glue” between them as shown in Figure 5-1. The glue was a predecessor of a lower-level EET model (see Figure 3-6) - a special-purpose language for specifying the linkage between user interface objects and interaction techniques on one side and application model abstractions on the other side. The user interface is edited using an existing interface building tool, SX/Tools [Kueh92], which supports designing custom objects in addition to providing predefined standard objects. The application model and the glue are edited in text editors under control of Interactive UIDE so that switching from design mode to run mode is instantaneous.

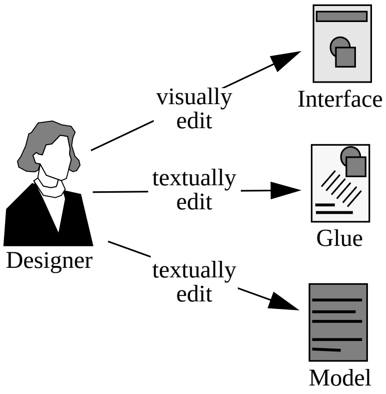


Figure 5-1: Design Mode

The designer can then ask the interview tool for advice, for suggestions and for help in inferring an interface or a model. In this mode, the designer reacts to questions and suggestions from the tool which changes the representations based on the designer's answers. This mode is shown in Figure 5-2.

The designer can instantly switch to run mode at any time. Figure 5-3 shows the system in run mode. At initialization time, Interactive UIDE's run-time component reads in the textual specifications of the model and the glue, and is linked to application-specific code if such code exists. The user interacts with the

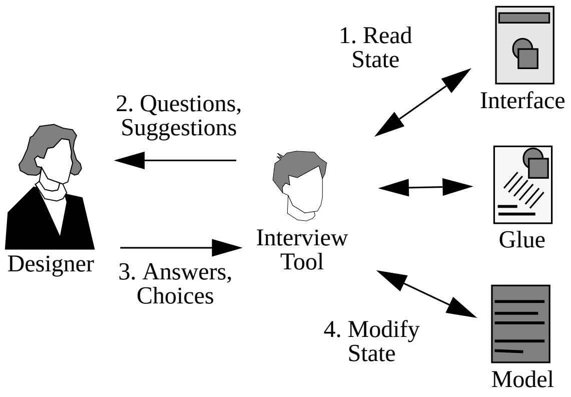


Figure 5-2: Interview Mode

application’s user interface which runs as a separate process under control of SX/Tools. Events which are relevant to the application are passed from the interface process to the run-time process which does computation and updates the user interface by sending events back to the user interface process.

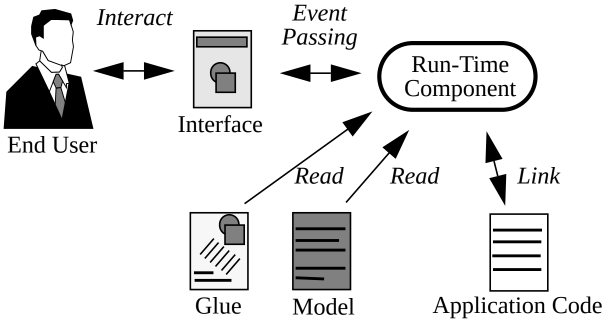


Figure 5-3: Run Mode

V.2.2 The Interview Component

The interview component can generate questions for building up or for improving the model or the interface. This component needs knowledge about application modelling, user interface design and their relationships. We separate this knowledge into question elements, or knowledge atoms, which encapsulate the information for a single question or suggestion presented to the designer. This knowledge structure is a combination of a rule base and a knowledge base which facilitates system-initiated questions.

There are two basic alternatives for the overall organization of these question elements. The first alternative is a graph-like structure, in which a question element explicitly encodes its follow-up questions. This provides for semantically meaningful sequences of related questions. However, the resulting graph structure is hard to understand and maintain. Figure 5-4 shows this knowledge base structure.

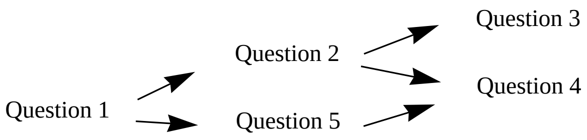


Figure 5-4: Graph Organization of Questions

The second alternative is a flat structure of questions with no provision for sequencing of related questions as shown in Figure 5-5. The question elements do not specify which question to ask next, so that the questions can be entered and evaluated independently, greatly simplifying the maintenance of the knowledge base.

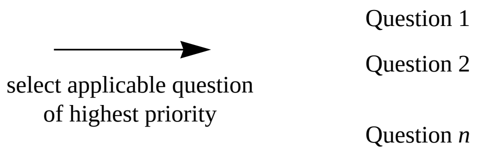


Figure 5-5: Flat Organization of Questions

However, there are situations where it is important to ask follow-up questions to a certain question. Imagine that there are two threads of questions about independent topics. It would be confusing if the presented questions would alternate between the topics. Therefore, we augment the flat structure with a simple mechanism to provide for explicit sequencing of questions by allowing a question to increase the priority of appropriate follow-up questions. This simple mechanism was sufficient for our moderately-sized rule base, but more sophisticated techniques from the Expert Systems field may be appropriate for larger rule bases.

Figure 5-6 shows one of the question elements. The applicability test determines if this question is relevant in the current context. The interview tool evaluates the applicable atoms and presents the candidate with the highest priority to the designer, using the parameterized question text. It also presents the atom-specific answers in addition to the generic answers available for all question elements such as “ignore question”, “ignore question permanently” and “help”. The designer selects one of these answers and the system executes the associated effect. The interview tool then re-evaluates the atoms and asks the next question if one applies.

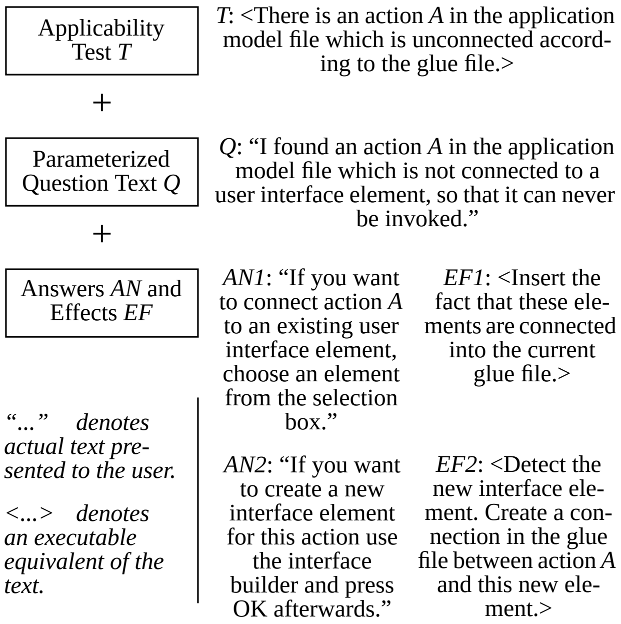

Applicability Test *T*
T: <There is an action *A* in the application model file which is unconnected according to the glue file.>
+
Parameterized Question Text *Q*
Q: “I found an action *A* in the application model file which is not connected to a user interface element, so that it can never be invoked.”
+
Answers *AN* and Effects *EF*
AN1: “If you want to connect action *A* to an existing user interface element, choose an element from the selection box.”
EF1: <Insert the fact that these elements are connected into the current glue file.>
AN2: “If you want to create a new interface element for this action use the interface builder and press OK afterwards.”
EF2: <Detect the new interface element. Create a connection in the glue file between action *A* and this new element.>
“...” denotes actual text presented to the user.
<...> denotes an executable equivalent of the text.

Figure 5-6: Internal Representation of a Question

V.2.3 Use of the Interview Tool

An interview component could provide support for both novice and expert users. However, the way in which they use the component will likely differ. Novice users will typically first build an interface by dragging elements from a palette and customizing them, and then invoke the interview tool to help them build the application behind the interface. Both of these activities require minimal training. Expert users will normally prefer to directly edit the representations instead of going through the interview process but may invoke the tool from time to time for design advice and consistency checking.

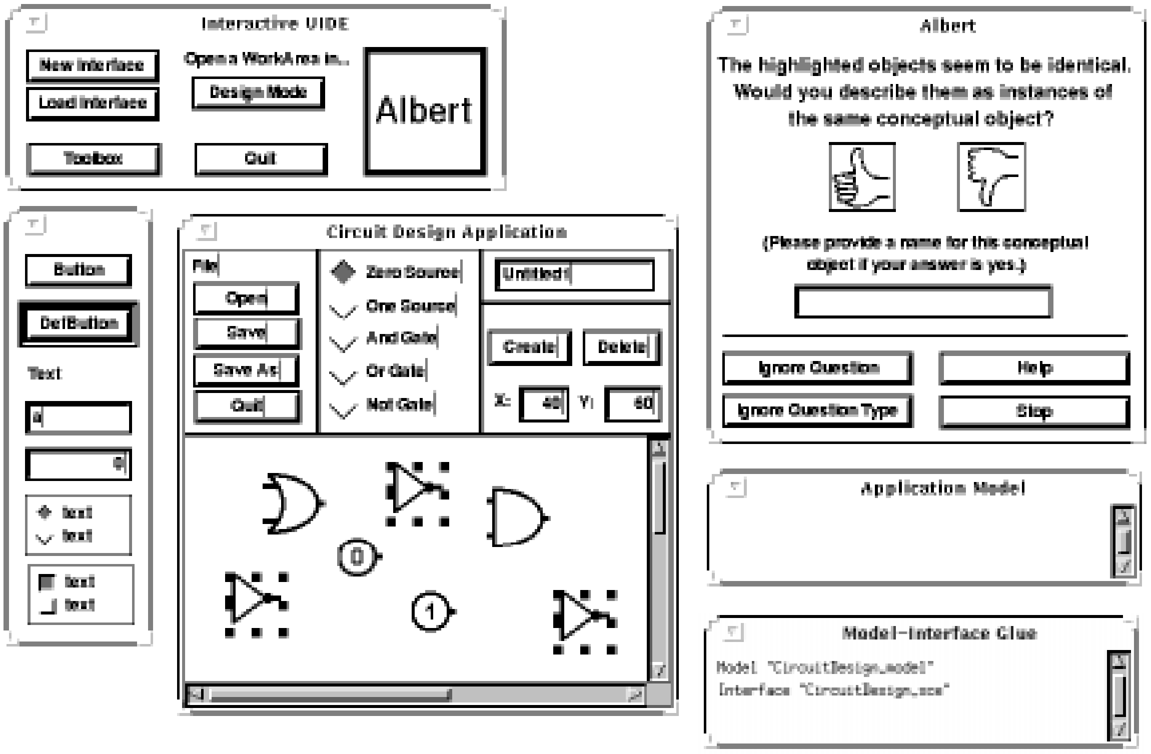
One of the advantages of this framework is the smooth transition from the novice to the expert level. Throughout the interview process, novice designers can watch the component change the textual representations in response to their answers. In this way, they can learn what information is needed and how answers are transformed into application model knowledge. Designers so inclined may then be able to edit the model directly after an initial training period.

V.2.3.1 Novice Use

We provide an extended novice example session in which a user interface and an application model for a circuit design application are constructed. The assumption is that the designer has no programming experience and no knowledge about application modelling so far. The designer does not have to edit the textual application model directly, nor does the designer have to understand the nature and use of the model at this point.

The designer first constructs the interface using the user interface builder. Figure 5-7 shows the main window of the combined interface and application model builder in the upper left-hand corner, titled “Interactive UIDE”. The designer has clicked on the “New Interface” button to start a new design and created some elements by dragging from element toolboxes. Figure 5-7 shows one of the toolboxes on the left (no title) and the user interface design in the center, titled “Circuit Design Application”.

The designer then invokes the intelligent component of the system by clicking on the “Albert” button.[74] This brings up two text editors, one for the application model and the other for the glue, both of which are empty so far besides the two



Albert

The highlighted objects seem to be identical. Would you describe them as instances of the same conceptual object?

(Please provide a name for this conceptual object if your answer is yes.)

Untitled1

X: 40 Y: 60

Model "CircuitDesign_model"
Interface "CircuitDesign_sce"

Figure 5-7: The Complete Framework of Interactive UIDE

lines in the Glue editor which specify which interface and which model are connected by this glue file. These are shown in the lower right-hand corner of Figure 5-7, titled “Application Model” and “Model-Interface Glue”. Finally, the interview tool computes the applicable question with the highest priority and presents it to the designer, shown in the upper right-hand corner, titled “Albert”.[74] In this case, the system has detected that several objects use the same bitmap and que-

---

[74] We originally called the interview component “Albert”. The screen shots still reflect this name because we cannot reproduce them to reflect the name change as the interview system is no longer operational.

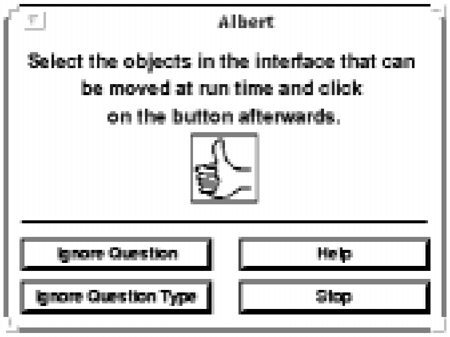
ries the designer if they are occurrences of the same concept. The highlighting of the affected user interface elements provides designers with the context of the question. Let us assume that the designer affirms and provides “NotGate” as the name for the object. The following initial application model is constructed.

```
Class NotGate
```

The glue representation is also changed to reflect that these three bitmap objects represent instances of the NotGate class. The next questions inquire about the other iconic objects one at a time (“Would you describe this interface object as an instance of a conceptual object?”). The model now looks like this.

```
Class NotGate
Class AndGate
Class OrGate
Class ZeroSource
Class OneSource
```

The system has limited knowledge about typical uses of iconic objects in applications, such as static icons for decoration purposes, icons which represent objects which can be accessed but not moved, and iconic objects which can be created, moved and deleted at run time. The question shown in Figure 5-8 encodes knowledge of this type. Assume the designer responds by selecting the circuit element objects and pressing the “thumbs up” button. The interview tool associates a position instance variable and a move action with the affected classes, so that the application model now consists of five classes with the following identical structure.



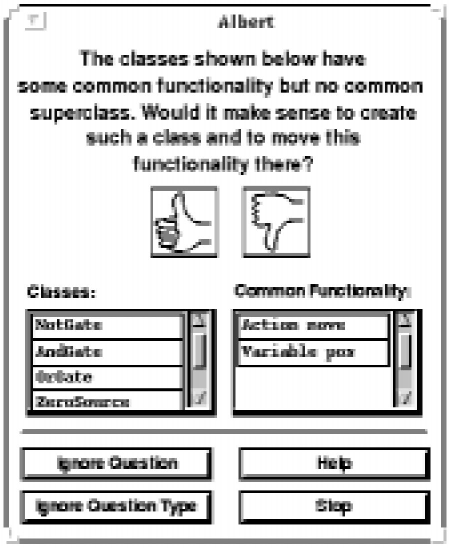
Albert

Select the objects in the interface that can be moved at run time and click on the button afterwards.

Figure 5-8: Knowledge About Common User Interface Behavior

```
Class X
     Position pos;
     Action move (Position newpos)
               Postcondition “pos := newpos”
```

The identical structure of these classes triggers another rule intended to help novice users structure their application model, which is shown in Figure 5-9.



Albert

The classes shown below have some common functionality but no common superclass. Would it make sense to create such a class and to move this functionality there?

Classes:

- NotGate
- AndGate
- OrGate
- ZeroSource

Common Functionality:

- Action move
- Variable pos

Figure 5-9: Structuring the Model

The designer confirms and gives “Element” as a name for that class. The model creates this class and moves the detected common functionality to it.

```
Class Element
     Position pos;
     Action move (Position newpos)
               Postcondition “pos := newpos”
Class NotGate SubclassOf Element
Class AndGate SubclassOf Element
Class OrGate SubclassOf Element
Class OneSource SubclassOf Element
Class ZeroSource SubclassOf Element
```

We will not further describe the interview process but it should have become clear how the interview tool can help novice designers build an application model from a user interface. So far, the interview tool cannot infer application sequencing so that the pre- and postconditions for sequencing have to be entered in textual form.[75]

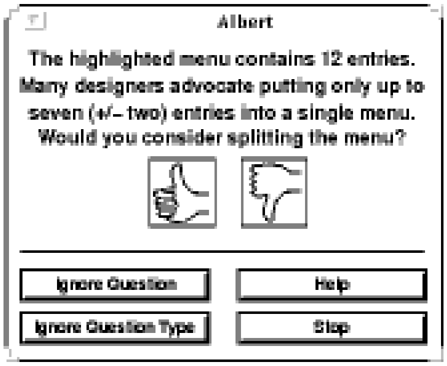
V.2.3.2 Expert Use

The previous section illustrated how novices can use the interview tool. Experts will usually invoke the tool in the role of a consultant only and will generally not rely on its initiative to build up the application model. The interview tool is purposefully redundant in the sense that a user interface and its corresponding model can be built using Interactive UIDE without invoking the interview component (using only the design and run modes but not the interview mode). While the

---

75. They could potentially also be demonstrated. However, we did not provide for a seamless integration of the interview tool with our demonstrational tools.

intelligent component is redundant in that sense, it may nevertheless be useful for an expert. Few designers are experts in both graphical user interface design and abstract application modelling. An interview tool may offer advice on both of these topics and can be used for consistency checking similar to automated spell checking. It may also be used as a “creativity agent”, a source of inspiration for user interface design. Figure 5-10 is a mock-up of a possible example of user interface design knowledge.



The highlighted menu contains 12 entries.
Many designers advocate putting only up to
seven (+/– two) entries into a single menu.
Would you consider splitting the menu?

Figure 5-10: Expert Use of the Interview Tool

V.2.4 Question Categories

The Interactive UIDE framework potentially provides for the generation of many questions and suggestions. We have grouped these questions into seven categories. One category builds up the model based on an existing interface, and, symmetrically, another category infers user interface elements from an application model. There could also be questions which improve a representation contextually or independently and questions which check for consistency. Table 5-1 summarizes these categories.

Table 5-1: Possible Question Categories

	Application Model	Glue	User Interface
Application Model Building	Modify	Modify	Read
User Interface Building	Read	Modify	Modify
Isolated Model Improvement	Modify	---	---
Isolated Interface Improvement	---	---	Modify
Contextual Model Improvement	Modify	Read	Read
Contextual Interface Improvement	Read	Read	Modify
Consistency Checking	Modify	Modify	Modify

V.2.4.1 Application Model Building

These questions build up a model from the user interface and from the interview process. Nearly all of the questions we have implemented fall into this category. The model is built by asking questions about the nature of all visible user interface elements one by one. Currently, there are no provisions for support on a higher level. The system should detect that the designer has put in similar information for the last two objects and ask a question at a meta-level like “select other objects which have similar behavior and click on the OK button”. It could then fill in model information for these objects at once, consequently reducing the number of repetitive questions. The system could also pose another meta-level question when the model inference process has just started, one which offers designers a choice of application model prototypes and asks which one is closest to the application they have in mind. It could then use this model and start asking from there, also reducing the overall number of questions.

V.2.4.2 User Interface Building

These questions may derive a user interface from an application model. This process is often referred to as interface “generation”. We avoid this term because it implies that this is a fully automated process with little or no options for the designer. The by-interview methodology facilitates a more interactive generation process, and could also generate new elements into an existing interface. For example, it could suggest putting a new operation in the same menu where the other operations on this object reside. In this way, user interfaces are not generated from scratch but can rather be updated incrementally as their underlying models evolve.

V.2.4.3 Isolated Model Improvement

In the process of building a model from the interface, there is a point where all the application information from the visual user interface has been exploited but where it is still desirable to improve the model. These questions suggest changes to the application model such as a restructuring of the class hierarchy. The question of Figure 5-9 is of this type.

V.2.4.4 Isolated Interface Improvement

It is also possible to refine the existing interface independent of the application model. For example, standard user interface design knowledge can be encoded in our questions so that this knowledge can be accessed and applied by people other than user interface designers. This is similar in spirit to ITS style rules [Wiec89] but augmented with the interactive interface to the knowledge. In this way, interface design knowledge is not only automatically applied, but the process is also visible and understandable to the designers so that they learn about interface design themselves while using the system. The question of Figure 5-10 is of this type.

V.2.4.5 Contextual Model Improvement

These questions suggest model changes based on interface properties. For example, the interview tool could detect that buttons or menu entries of certain actions are grouped together visually and ask the designer if this represents a semantic grouping that should be captured in the application model.

V.2.4.6 Contextual Interface Improvement

The interview tool could suggest improvements to an interface even if the interface is already complete and consistent with the application model. For example, assume that the actions on a certain type of object are all available in the current interface, but that they reside in different locations. The system could issue a warning about this design and suggest moving them to one location, such as an object-specific popup menu.

V.2.4.7 Consistency Checking

Finally, there is one category of questions concerned with maintaining the consistency of the interface and the application model such as “The application model contains the action Align for class VisualElement which currently cannot be invoked from the interface. Is this intentional?” or “A connection in the glue file refers to a non-existent application action. Do you want me to delete the connection?”.

V.2.5 Discussion

We attempt to classify our tool in the context of the established approaches for building user interfaces.

The interview tool encodes domain-specific knowledge in its questions, but it is different from knowledge bases in that the initiative is with the tool rather than with the user. The user invokes the tool but it is the tool that queries the user in order to extract knowledge about design decisions, a system-driven knowledge acquisition process.

The interview tool asks natural language questions, but the questions it asks are canned. It has no capabilities in natural language generation or understanding. Instead, the designer provides answers through selecting interface objects and filling out forms.

When used to infer an application model, our system starts from a user interface example but it is not a by-example approach such as Peridot [Myer88] in a strict sense because a user interface example alone is not sufficient for inferring a semantic application model. However, our approach still shares a similar philosophy. Examples are inherently easier to understand than abstract concepts, and our system makes use of this fact.

Our system is not a with-example approach either. In Myers' and Halbert's definition [Myer88], programming with example is a generalized macro recording approach, following the philosophy of "do what I did" rather than the "do what I mean" philosophy of by-example programming. The interview tool does not follow this approach for inferring the structure of an application because this would translate into demonstrating this static structure in a temporal dimension.

This concludes our discussion of the interview component. As mentioned in the introduction to this chapter, we do not believe that an interview approach can ever completely cover any interesting domain. However, we feel that interview tools can be genuinely helpful during any design activity by making prior domain knowledge executable. We further believe that the key to the success of such components is to (a) always keep their use strictly optional by also providing for alternative manual entry and to (b) never change a part of the representation "behind the users' back", such as changing text that is located off-screen without their explicit consent.

CHAPTER VI

USABILITY RESULTS

We performed a series of informal usability tests on our demonstrational tools as they evolved, and we also conducted two more formal studies on which we will report in this chapter.

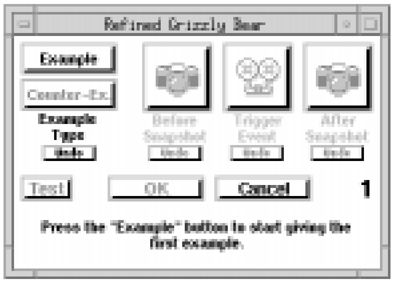
Each of these two studies involved ten subjects. One group of subjects was recruited from Georgia Tech sophomores and juniors taking a required introductory class in psychology (the “non-programmer study”). The other group consisted of computer science faculty and graduate students (the “programmer study”).

VI.1 Testing on Non-Programmers

This section describes an experiment we performed on subjects with little or no programming experience.

VI.1.1 Further Refinements to Grizzly Bear’s User Interface

In this experiment, all subjects used Grizzly Bear. We made some improvements to Grizzly Bear’s user interface before this study was conducted, based on feedback from the earlier “programmer study” (Section VI.2). Figure 6-1 shows Grizzly Bear’s control panel as used in this experiment. It contains the following changes compared to the control panel discussed in Section IV.3.1 (there are no differences in the inferencing).



Press the "Example" button to start giving the first example.

Figure 6-1: Grizzly Bear's Refined User Interface

First, we improved Grizzly Bear's status messages to provide more guidance to the user, and to provide feedback in a more human-readable form. For example, the status message after recording an event now displays something like "What should happen when 'circle1' is 'pressed'?" rather than just providing a raw print-out of the triggering event.

We were originally driven by implementation considerations when adopting the term "negative example" for an example which shows when behavior is *not* supposed to be triggered. We changed the name to "counter-example", and also made the "Example Type" decision the first step in giving an example rather than an intermediate step because subjects in the programmer study were sometimes confused about which type of example they were currently giving. We also highlight the background of the "Example Type" buttons for the duration of an example to provide additional context (the “Example” button highlights in light green, the “Counter-Example” button highlights in orange).

A major improvement in Grizzly Bear’s usability were the “undo” buttons shown in Figure 6-1. In the earlier version, mistakenly providing a wrong snapshot in, say, the fourth example meant that one had to give the entire demonstration again from scratch. The new “undo” buttons let designers undo one or more steps of a demonstration at any time (one normally goes through the iconic buttons from left to right - the “undo” buttons let one go back from right to left).

We also eliminated the check-boxes for ignoring certain types of events (see Figure 4-2). In our experience, subjects rarely used them because they did not understand the nature of the events at that fine-grained level. The “Ignore Enter Events” check-box was not needed in the first place, and we could replace the “Ignore Motion Events” by automatically ignoring motion events unless a significant number occurred (thus providing an alternative mechanism for suppressing accidental motion events).

Finally, we made testing the demonstrated behavior easier by putting a “Test” button into Grizzly’s interface, displayed the ordinal number of the current example, and provided a warning dialog if the designer has already demonstrated behavior and then presses the “Cancel” button.

VI.1.2 How Subjects Were Recruited

The subjects for this study were recruited through a sign-up sheet in front of the classroom for a required introductory psychology class at Georgia Tech. The subjects were offered $20 for participating.

The ten subjects that actually participated in the experiment happened to come from ten different academic disciplines, and they were all sophomores or juniors. Two of the subjects had no prior programming experience at all, seven had only had experience in the form of required introductory programming classes, and one had more substantial programming experience from a former co-op job (see Table E-1 for the detailed demographics).

VI.1.3 Experimental Procedure

The subjects first filled out a questionnaire (Appendix E.1). They then proceeded at their own pace for sixty minutes following written instructions (Appendix E.2). The experimenter only broke in if the experimental prototype failed. In such a case, the experimenter would halt the clock, bring the system back to the state just before the failure occurred, and restart the clock.

The written handout consisted of eleven pages. The first page contained general information about the experiment as well as instructions on how to bring the system back to the initial state (which is done after each task). The second page contained a high-level description of Grizzly Bear. Pages three through seven contained detailed step-by-step instructions for three training tasks. Training task A consisted of making a red circle become blue when it is clicked (one example), training task B was to make a circle jump to wherever one clicks (two examples), and training task C was to make the circles in a window change their color to blue when clicked while making sure that rectangles do not exhibit this behavior (one example and one counter-example). There is no “thinking” involved in following these step-by-step instructions so far. Indeed, all subjects were able to complete the instructions to this point, which took about twenty-five minutes on average. The remaining four pages of the handout each described a task that the subject was to accomplish. Subjects were allowed to skip a task in favor of trying another one; however, no subject actually chose to do so.

The first task was to make a rectangle change its color from yellow to blue when clicked. This introductory task was designed to be easy; it involved no more than replicating the demonstration in training task A in a slightly different context. Indeed, all subjects could complete task 1, which took about two minutes on average.

The second task was to have red circles be deleted when clicked while nothing happens to yellow circles. Task 2 is similar in spirit to training task C. However, the demonstration required three examples overall (two examples and one counter-example), and thus required a higher intellectual effort than the first task. Nine out of ten subjects could accomplish this task[76], it took six minutes on average.

The third task was to create a new circle whenever one clicks in a window. It was the first task to involve the creation of objects; it is otherwise similar to training task B. Seven out of ten subjects could complete the task, in an average time of five and a half minutes.

The last task was intentionally designed to be much harder than the others. It involved making all currently-red circles green when an on-screen button is pushed. The task could not be accomplished without setting up a different *before* snapshot for at least one example (at least two examples were needed overall). However, the possibility of setting up differently for different examples was only mentioned in passing in the introduction of Grizzly Bear, and none of the training tasks involved doing so. Thus, the subjects had to discover a completely new type of demonstration, rather than simply repeating or extending the demonstrations in the training tasks. Three of the ten subjects could complete this task, it took them about fifteen minutes on average.

---

76. Note that "not accomplishing the task" means that either (a) the subject did not understand how to demonstrate it to Grizzly Bear or (b) the subject ran out of time while trying to accomplish the task or (c) the subject ran out of time before getting to this task.

Table 6-1 presents a summary of the data gathered. (Table E-1 in the appendices contains the complete data.)

Table 6-1: Summary Data from the Non-Programmer Study

	Min.	Max.	Mean	Median	Std. Dev.
Tasks Completed	1	4	2.90	3.00	0.99
Time Needed for Instructions	17:47	53:06	25:21	21:22	10:31
Task 1	1:26	4:43	2:12	1:59	0:58
Task 2	2:45	14:24	6:03	5:18	3:37
Task 3	3:11	7:09	5:36	6:38	1:55
Task 4	9:53	19:32	15:24	16:47	4:58

VI.1.4 Analysis

The most important result from this study is that subjects with little or no prior programming experience can indeed “program” some user interface behavior using Grizzly Bear within an hour, given only written instructions.

However, we also wanted to test our hypothesis that subjects with more programming experience[77] will do better with Grizzly Bear. We tested the following more exact hypothesis.

---

[77] We measured programming expertise by asking the subjects how many hours per week they programmed during their most intense week of programming. We will thus refer to “prior intensity of computer programming” in our hypotheses rather than to the more general term “programming experience” which is harder to quantify.

"*There is a positive correlation between prior intensity of computer programming and the number of tasks accomplished.*[78]"

Working with a 5% significance level and assuming a normal distribution for both variables, there is no statistically significant positive correlation between prior programming intensity and the number of tasks subjects could accomplish with Grizzly Bear.[79] This surprised us, especially because the following hypothesis holds.

"*There is a positive correlation between prior intensity of computer use and the number of tasks accomplished.*[80]"

The correlation is statistically significant based on the same significance level and assumptions.[81]

We are encouraged by these results, which seem to indicate that solid computer literacy is the only requirement for defining simple behavior with Grizzly Bear, and that prior programming experience makes no significant difference.

---

[78] The two variables are contained in rows eight and ten of Table E-1.

[79] The correlation is $r=0.117$, well below the critical value for significance $r>0.549$. (The table of critical values we used is from [Koos85], page 275.)

[80] The two variables are contained in rows five and ten of Table E-1.

[81] The correlation is $r=0.625$ (greater than the critical value $r>0.549$).

VI.2 Testing on Programmers

This section describes an experiment we performed on subjects with substantial programming experience. While the primary purpose of our demonstrational tools is in supporting an audience with little or no programming experience, we wanted to make sure that our environment is also of value for programmers.

VI.2.1 How Subjects Were Recruited

All subjects were computer science faculty or graduate students. They volunteered for a three-hour experiment (on a “friendship basis”, there was no compensation). Some subjects had previously seen me give talks or demos concerning Inference Bear while others had not (as listed in Table F-1 for each subject).

VI.2.2 Experimental Procedure

There were four groups of subjects which all performed the same sequence of tasks. The first group used Inference Bear, Expression Finder, and textual editing. A second group used Grizzly Bear and textual editing. A third group exclusively used Grizzly Bear (they were allowed to read its textual output if they wanted to, but they were expressly forbidden to modify it). The final group programmed the behavior textually in the Elements, Events & Transitions language.

All subjects were asked to accomplish the same eight tasks. The first four were isolated tasks similar to the tasks in the “non-programmer” study. The remaining four tasks were to define the behavior of the “mini-editor” described in Section IV.3.4.

The experimental procedure was much less thorough than in the “non-programmer study”. We acquainted the subjects with the environment by informally showing them examples of how to use the demonstrational tools, or how to use the Elements, Events & Transitions language.[82] We orally described what the next task was after each task (there were no written instructions).

Finally, we would break in if we sensed that the subject felt stuck or got overly frustrated. While we stopped the clock when we broke in, a task timing for a subject that received a hint is of course not comparable to a task timing of a subject that did not. We have not aggregated the task timing results for that reason. Table F-1 shows tasks for which a subject received help in bold, and shows the nature of the hint in a footnote.

VI.2.3 Analysis

We have not performed any formal statistical analysis for this study because of the small sample size (only two or three subjects per group) and

---

[82] It is worth noting that we spent much more time discussing the language before an experiment than discussing the demonstrational tools (an estimated half an hour for the language versus an estimated ten minutes for the demonstrational tools).

because of the flaws in the experimental procedure described earlier. However, we want to relate some behavioral observations.

Part of the original motivation for this study was to find out if subjects had an easier time using two separate demonstrational tools (Inference Bear and Expression Finder) or a single albeit more complex demonstrational tool (Grizzly Bear). From our observations, we feel that a single tool is preferable because subjects were often confused about which tool to call (they would sometimes start demonstrating before they realized they called the wrong tool), and because the additional complexity of Grizzly Bear over Inference Bear is hidden until one clicks the “Negative Example” button (the “Counter-Example” button in the refined version of Grizzly Bear presented in Section VI.1.1).

Another result of this study is that subjects who are able to understand the generated textual language (“programmers”) made it very clear that they disliked being kept from editing it.

Another rationale for this study was to compare subjects that had to use the textual language (group four of Section VI.2.2) to subjects that had to demonstrate the behavior (group three), and to subjects that were free to do either (group two). For the particular tasks of this study we must conclude that textual programming is the fastest alternative for subjects that are able and willing to learn the formal language beforehand.[83] However, it is also interesting that the subjects who were free to choose typically first demonstrated a rough version of the desired behavior and then went from there textually, rather than trying to do everything textually.[84] Some subjects also communicated verbally after the experiment that they like the idea of demonstrating a rough version of a behavior (so that they would not have to start from a blank slate).[85]

VI.3 Discussion

Our “non-programmer study” seems to indicate that a non-programming audience can indeed use Grizzly Bear to define user interface behavior in less than sixty minutes, and that programming experience does not play a statistically significant role in being able to use Grizzly Bear.

Our “programmer study” could not demonstrate an advantage in speed over textual programming for a programming audience. However, we believe, based on the anecdotal evidence, that there can be a role for demonstrational

---

[83] This observation does of course not apply to the “non-programmer” study as its subjects would likely not be able to learn and use a textual programming language within sixty minutes.

[84] More precisely, out of the twelve tasks concerning the “mini-finder” for the three subjects using Grizzly Bear or textual programming, the subjects used pure textual programming in two cases, pure demonstrational programming in three cases, and a combination in seven cases.

[85] One of the subjects remarked literally: “I’ve never used a demonstrational system before. For the first time I am realizing how useful it can be.”

tools in professional programming when properly combined with a textual environment.

CHAPTER VII

CONCLUSION

The previous chapters explain the Elements, Events & Transitions model and its interactive tools in great detail. This chapter concludes with a more general discussion of what we have learned. It also lists our main contributions, and discusses possible extensions and future research directions.

VII.1 Discussion

This section will present some of the lessons that we have learned in the course of this research. We will discuss these issues from a more general perspective, in contrast to the discussion sections that conclude the individual thesis chapters.

VII.1.1 On the Optimal Complexity of Modelling Languages

For the purpose of this section, we will define a “modelling language” as a special-purpose user interface language that directly or indirectly controls run-time behavior. We have argued earlier that there is a need for such languages if non-programmers are to participate in defining and modifying user interface behavior (Section III.1.1).

If this is so, the question arises how many modelling languages should be made available, and how many levels of abstraction these modelling languages should provide.

The answer naturally depends on the purpose of the modelling. We will characterize the possible combinations of these factors in this section. Table 7-1 lists all possible combinations.

Table 7-1: Specifying the Human-Computer Dialog

	No modelling language	One modelling language	Many modelling languages
Single level of abstraction	A simple conventional implementation using a general-purpose programming language.	An environment using a single special-purpose language for user interface behavior (typically at a low level of abstraction).[a]	(We are not aware of any user interface design tools taking this approach, as it seems to unnecessarily burden its users.)
Multiple levels of abstraction	A conventional implementation that internally uses multiple levels of abstraction to structure the application code.	An environment where the same language is used to model behavior at several abstraction levels.[a]	A full-fledged model-based environment which captures knowledge on many aspects of the design.

a. These are the possible uses of the EET model described in this thesis.

VII.1.1.1 No Modelling Language, Single Level of Abstraction

Most user interface implementations use a general-purpose programming language to describe the human-computer dialog at a single level of abstraction.

This seems appropriate if all of the user interface behavior is implemented by programmers, and if the application is written for a single platform.

VII.1.1.2 No Modelling Language, Multiple Levels of Abstraction

Some implementations of graphical applications use a general-purpose programming language to internally describe the human-computer dialog at multiple levels of abstraction. For example, part of the higher-level dialog specification may read “the user then supplies the file to be printed and the name of the printer”. A lower-level dialog specification then prescribes that, say, the file is chosen through a selection box while the printer is chosen by clicking on an iconic representation. (Both these specifications are in the form of general-purpose programming language code.)

Separating the dialog description into multiple levels of abstraction seems appropriate for applications that run on multiple computing or windowing platforms. While it still takes programmers to describe any aspect of the dialog, this approach avoids having to re-implement all of the dialog for every new platform.

VII.1.1.3 One Modelling Language, Single Level of Abstraction

Many user interface building tools let their users describe behavior in an interpreted scripting language. For example, Microsoft’s Visual Basic environment includes a user interface building tool that is well integrated with such a language (Basic), Apple Computer’s HyperCard environment uses the same approach (HyperTalk), and so do many user interface builders (by e.g. providing an interpreter for a subset of the C programming language[86]). Our Elements, Events &

---

[86] For example, the SX/Tools interface builder takes this approach [Kueh92].

Transitions language can also serve as such a scripting language when used at a single level of abstraction (as was done throughout Chapter IV).

This approach seems most appropriate for rapidly prototyping user interfaces. It also imposes the least learning overhead of the alternatives discussed in this section (assuming that the users do not already know any of the modelling or programming languages beforehand, of course). For those reasons, we favor this approach towards letting non-programmers describe user interface behavior.

VII.1.1.4 One Modelling Language, Multiple Levels of Abstraction

Assuming that the modelling language does not contain vocabulary that is specific to a particular level, it can be used to describe the dialog at multiple levels of abstractions. For example, separate Elements, Events & Transitions models can describe the dialog at different abstraction levels (see Section III.4).

The motivation for this approach is a more modular design, similar to the motivation for using a general-purpose programming language in the same manner. In addition, using the same special-purpose language that is also used for low-level user interface behavior is potentially more accessible to user interface designers than a general-purpose programming language.

We speculate that our demonstrational tools can help designers construct higher-level models in a roundabout, indirect fashion. They first expose the designers to the language in their own domain, namely concrete user interface behavior. Over time, designers so inclined can then learn to directly author behavior textually at the user interface level, and possibly at a higher level of abstraction afterwards.[87]

VII.1.1.5 Many Modelling Languages, Single Level of Abstraction

Using several special-purpose languages to describe behavior at the same level of abstraction seems to unnecessarily complicate the design process, and is mentioned here only for completeness. We are not aware of any user interface tools using this approach.

VII.1.1.6 Many Modelling Languages, Multiple Levels of Abstraction

In this approach, the user interface design environment supplies a number of modelling languages that are each geared towards describing dialog-related information at a particular level of abstraction [Suka93, Szek95]. For example, there can be a special-purpose language for describing the tasks that the user is accomplishing, another language for describing the interaction techniques used, a language for describing the characteristics of the computing platform, and so on.

This approach has the benefit of providing languages that are well-suited for their purpose, and can capture more declarative knowledge than any other

---

87. We must caution that letting designers describe user interface behavior at multiple levels of abstraction was not a major focus of this work, and that this “indirect learning effect” is our educated speculation rather than an observed matter of fact.

approach. It seems most appropriate for large user interface designs, and when the knowledge captured by the modelling languages is used for other purposes in addition to being used for run-time control (see Section II.2).

VII.1.2 The Influence of the Event Structure on PBD Systems

Most modern windowing systems treat user input as a stream of events. The structure of these events has a significant impact on the inferencing power and ease-of-use of a Programming By Demonstration (PBD) system.

For the purposes of this section, what we call the “event structure” is the definition of the types of events that come in from the windowing system, including their parameters (e.g. “the Properties window was entered while the shift key was down”).[88]

It is obviously important that an event to which the user wants to tie behavior is indeed provided by the system. However, it is also important that this event can easily be told apart from others. For example, assume that the designer wants a “trashcan” icon to become highlighted when entered with the mouse. Ideally, the event will explicitly state that the event occurred on the “trashcan” object, and that the event that occurred was an “entered” event. This way, the PBD system can generate output that reads “if the ‘trashcan’ is ‘entered’ then...”.

---

[88] This section will not discuss the potential benefits of placing high-level events into the input stream (e.g. “file selection task complete”), which is the topic of a separate dissertation [Kosb94].

Contrast this with an event that e.g. states “mouse was moved from (123,37) to (129,43)”. In this case, telling the “trashcan entered” event from the other mouse movement events involves complex computation (thus requiring an impracticably large number of examples if this behavior is to be inferred by a domain-independent PBD system).

The event parameters are also important. Suppose that a designer wants to demonstrate that “shift-clicking” an object will select this object without deselecting others. Ideally, the “click” event will carry status information about the shift key. In that case, the user can directly state that “if an object is clicked while the shift key is held down then...”.

Contrast this with a situation where events do not carry modifier key information. In that case, the designers have to keep track of the shift key status themselves. This can for example be done by introducing a boolean variable *flag*, and by demonstrating that pressing the shift key sets *flag* to true while releasing this key sets *flag* to false. Only then can they state “if an object is clicked while *flag* is true...”, making the overall demonstration of this behavior much more complex.

In conclusion, while the *reasoning* of a domain-independent PBD system may be independent of a particular event structure, its actual *usefulness* is not.

VII.1.3 Textual Languages as the Basis of PBD Systems

Programming By Demonstration (PBD) systems intended for a non-programming audience often include a graphical language for the representation of demonstrated behavior, like e.g. Pygmalion and Mondrian (both in [Cyph93]) and KidSim [Cyph95]. We have deliberately put all our effort into an inferencing mechanism at the expense of a visual language (primarily because another student at our institution has in turn put all her effort into a visual language for PBD systems, see Appendix B).

Nothing precludes the use of our demonstrational systems with a visual language equivalent to our Elements, Events & Transitions language. But is there a place for PBD systems that indeed present their output in a textual language?

We believe that this is the case, especially if the target audience are "power users" - users that may not have any experience in a general-purpose programming language but are familiar with e.g. the macro language of a spreadsheet application. Their motivation for using a PBD system is typically not empowerment, as they are also capable of crafting the equivalent textual specification. Instead, they may want to use a PBD system from time to time if it is easier or faster to demonstrate a piece of behavior rather than to write it from scratch.

We also believe that a textual programming language can sometimes be suitable even for a non-programming audience if the language is specific to their needs (as is also argued in [Nard93], especially in its third chapter). Apple Computer's HyperTalk language (part of the HyperCard prototyping environment) is a good example of a successful textual behavior specification language for end-users.

VII.2 Contributions

The thesis contributes to the state of the art in three dimensions. First, its specification language, the Elements, Events & Transitions model, is the first user-level language for interface behavior explicitly designed to be used with demonstrational tools.[89] Second, its demonstrational tools, most notably Grizzly Bear, cover an unusually wide spectrum of user interface behavior, and are unique in keeping their reasoning independent of the characteristics of any particular user interface toolkit. Finally, the thesis is the first to explore in depth how to best combine the ease-of-use of the demonstrational approach with the expressive power of the model-based approach.

VII.3 Implementation Issues

This section discusses some desirable improvements to our current prototype implementation from an engineering perspective.

---

89. It is primarily the simplicity and uniformity of the EET language that makes it suitable for generation by demonstrational tools. There are only three statements for manipulating elements, which can all be produced by Grizzly Bear (simplicity). Using the same “set expressions” for invoking transitions and within statements means that the reasoning mechanism for set expressions can be used for all (uniformity). Finally, the pseudo-parallelism discussed in Section III.5.1 supports our snapshot-based PBD approach well.

The user interface builder that is the basis for our prototype implementation, SX/Tools [Kueh92], runs as a separate process from its client code, requiring inter-process communication to access and change the state of its user interface elements. This turned out to be a major performance bottleneck in the interpretation of Elements, Events & Transition (EET) models. In addition, SX/Tools does not have the ability to send a notification whenever an aspect of the user interface is edited by the designer, which required us to indeed take complete snapshots during demonstrations when we could otherwise just have tracked changes (see Section IV.4.1.1). Finally, SX/Tools is a commercial product that we cannot distribute freely to other researchers. For these reasons, we intend to move the EET interpreter and Grizzly Bear to a new platform.

As discussed earlier, other worthwhile implementation improvements are building a smarter EET interpreter (Section III.5.2.1) and providing an EET compiler (Section III.5.2.2).

VII.4 Extensions

This section will discuss some desirable extensions to an EET-based user interface design environment that do not require substantial new research.

The most obvious extension to this research is to present the output of demonstrations in an editable visual language (similar to the graphical rewrite rules used in KidSim [Cyph95]). Appendix B presents a visual language that we feel is especially well-suited for integration with Grizzly Bear.

Another desirable extension to the design environment is a single-stepping tool for debugging purposes. It would enable the designer to inspect which transitions fire while interacting with the design in test-drive mode. For example, this tool could highlight the text of the statement that is currently executed.

Finally, the Elements, Events & Transitions model provides for multi-valued attributes (Section III.2.1) but Grizzly Bear's current inference types only supply reasoning for single-valued attributes (Section IV.4.1.2). This has not been a major limitation on the power of our demonstrational tools because marking elements with user-defined single-valued attributes can often substitute for keeping a list of element names in a multi-valued attribute. Nevertheless, we would like to extend Grizzly Bear's reasoning to multi-valued attributes. This should be possible by simply augmenting the existing inference types with additional code that tests multi-valued target variables for list operations such as insertion and deletion.

VII.5 Future Work

As the dissertation title implies, combining model-based user interface design and Programming By Demonstration (PBD) was a major focus of this work. Throughout most of this thesis, we have used our Elements, Events & Transitions model to capture behavior at a low level of abstraction, namely the widget manipulation level. This has enabled us to build demonstrational tools that naturally map demonstrations one-to-one to constructs of our model. An issue that we have not addressed is how to use programming by demonstration to input high-level model constructs. For example, a higher-level model may specify that commands *throughout the interface* are invoked first, before their arguments are supplied (pre-fix), or that their parameters are supplied first (post-fix) [Kova92]. It is an issue of future research if the definition of behavior at such an abstract level lends itself to a PBD approach. To this end, the designer would have to either manipulate abstract objects, or to give demonstrations at the user interface level that the PBD system then generalizes to a much more abstract level.

VII.6 Conclusion

Designing executable graphical user interfaces is only one instance of a fundamental remaining problem in human-computer interaction, namely how to make *programming computers* accessible to everybody, rather than just *using computers*. Future work towards this end includes more research on domain-oriented specification languages, on visual languages, on programming by demonstration, and on visual debugging tools. We speculate that more and more of what is “programming” today will in the future be done by people that would not describe themselves as “programmers”. We hope that this research makes at least a modest contribution towards this vision.

APPENDIX A

EET Language Definition

This appendix precisely specifies the syntax of the Elements, Events & Transitions (EET) language in Backus-Naur Form (BNF). It complements the informal language description of Chapter III, and may also be useful for a future reimplementation of the language.

The following three regular expressions define tokens of the language that are used throughout the BNF definition later.

```
cardinal       [0-9]+
name           [a-zA-Z][a-zA-Z0-9_\-]*
quotedstring   \”[^”]*\”
```

That is, a “cardinal” consists of one or more digits, a “name” is a letter followed by any sequence of other letters, digits, underscores or hyphens, and a “quotedstring” consists of any sequence of characters enclosed in quotes.

The rest of this appendix contains the language definition. Note that characters enclosed in single quotes, such as ‘Transition’ or ‘<=’, represent the verbatim occurrence of the enclosed character sequence.

```
eetModel ::=
      /*empty*/ |
      eetConstruct eetModel

eetConstruct ::=
      eetPrototype |
      eetObject |
      eetTransition

eetPrototype ::=
      ‘Prototype’ eetOptionalColon name eetProtoOrObjBody

eetObject ::=
      ‘Object’ eetOptionalColon name eetProtoOrObjBody

eetProtoOrObjBody ::=
      ‘{’ eetAttributes ‘}’ eetOptionalSemicolon

eetAttributes ::=
      /*empty*/ |
      eetAttribute eetAttributes

eetOptionalType ::=
      /*empty*/ |
      ‘<’ name ‘>’

eetAttribute ::=
      eetOptionalType name eetAttributeValues ‘;’

eetAttributeValues ::=
      /*empty*/ |
      eetAttributeValue eetAttributeValues

eetAttributeValue ::=
      name |
      cardinal |
      quotedstring

eetOptionalColon ::=
      /*empty*/ |
      ‘:’

eetOptionalSemicolon ::=
      /*empty*/ |
      ‘;’

eetTransition ::=
      ‘Transition’ eetTransExpr
                  ‘(’ eetTransitionPars ‘)’
                  ‘{’ eetStatements ‘}’

eetTransitionPars ::=
      /*empty*/ |
      eetTransitionParsAux

eetTransitionParsAux ::=
      eetTransitionPar |
      eetTransitionPars ‘,’ eetTransitionPar
```

```
eetTransParType ::=
      name

eetTransParOptionalConstraint ::=
      /*empty*/ |
      eetComparison eetExpr

eetTransitionPar ::=
      eetTransParType name eetTransOptionalConstraint

eetDeleteStmt ::=
      ‘delete’ eetSetExpr

eetCreateStmt ::=
      ‘element’ name ‘:=’ ‘create’ eetSetExpr

eetAssignment ::=
      eetAttributeExpr ‘:=’ eetRhs

eetListDeleteStmt ::=
      eetAttributeExpr ‘-=’ eetRhs

eetListAddStmt ::=
      eetAttributeExpr ‘+=’ eetRhs

eetListClearStmt ::=
      eetAttributeExpr ‘:=’

eetOptThrowReceiver ::=
      /*empty*/ | name ‘::’

eetThrowStmt ::=
      ‘throw’ eetOptThrowReceiver eetSetExpr ‘.’ name
            ‘(’ eetThrowPars ‘)’

eetStatementNoSemi ::=
      eetDeleteStmt |
      eetCreateStmt |
      eetAssignment |
      eetThrowStmt |
      eetListAddStmt |
      eetListDeleteStmt |
      eetListClearStmt

eetStatement ::=
      eetStatementNoSemi ‘;’

eetStatements ::=
      /*empty*/ |
      eetStatement eetStatements
```

```
eetAttributeExpr ::=
      eetSetExpr eetDottedNames

eetTransExpr ::=
      eetSetExpr '.' name

eetRhs ::=
      eetExpr

eetExpr ::=
      '(' eetExpr ')' |
      eetExpr '+' eetExpr |
      eetExpr '-' eetExpr |
      eetExpr '*' eetExpr |
      eetExpr '/' eetExpr |
      '-' eetExpr |
      eetIdentifier

eetSetExpr ::=
      name |
      '*' |
      '(' eetSetExprSub ')'

eetSetExprSub ::=
      '(' eetSetExprSub ')' |
      eetSetExprSub '&&' eetSetExprSub |
      eetSetExprSub '||' eetSetExprSub |
      '!' eetSetExprSub |
      eetIdentifier eetComparison eetExpr

eetNumberId ::=
      cardinal

eetStringId ::=
      quotedstring

eetParamId ::=
      name

eetSlotId ::=
      eetSetExpr eetDottedNames

eetListLengthId ::=
      '|' eetSlotId '|'

eetIdentifier ::=
      eetStringId |
      eetNumberId |
      eetParamId |
      eetSlotId |
      eetListLengthId

eetThrowParType:
      name

eetThrowPar ::=
      eetThrowParType name ‘:=’ eetRhs

eetThrowPars ::=
      /*empty*/ |
      eetThrowParsAux

eetThrowParsAux ::=
      eetThrowPar |
      eetThrowPars ‘,’ eetThrowPar

eetComparison ::=
      ‘==’ |
      ‘!=’ |
      ‘<’ |
      ‘<=’ |
      ‘>’ |
      ‘>=’

eetDottedNames ::=
      ‘.’ eetSpec |
      eetDottedNames ‘.’ eetSpec

eetStringSpec ::=
      name

eetListSpec ::=
      name ‘[’ cardinal ‘]’

eetSpec ::=
      eetListSpec |
      eetStringSpec
```

APPENDIX B

A Visual Language for Grizzly Bear

Erica Sadun is developing a new visual specification language for user interface behavior [Sadu96]. Figure B-1 contains an example specification.

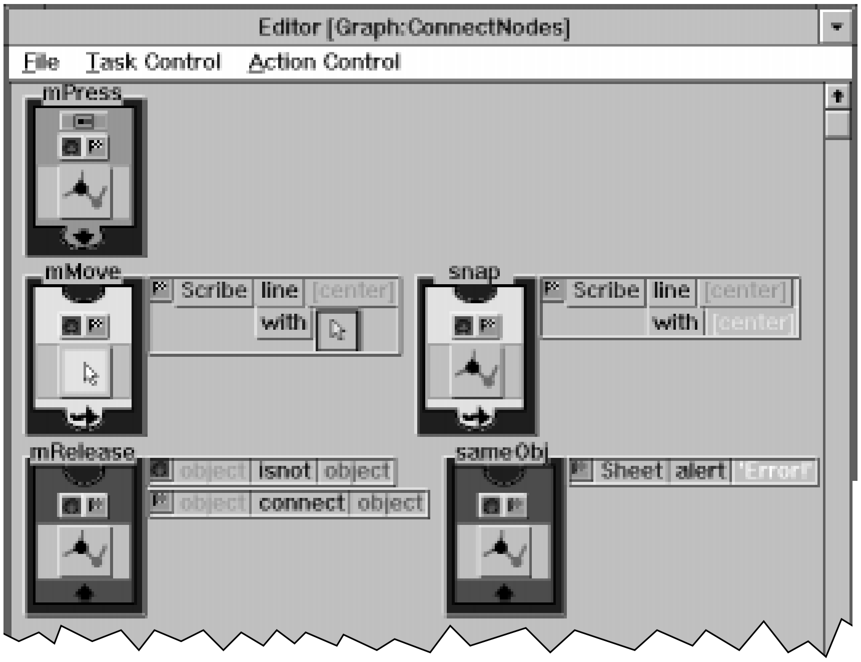


Figure B-1: A Visual Notation for User Interface Behavior

We will not explain this specific visual language in detail here (the screen represents part of the dialog description for a graph editor), but note that the three graphical expressions on the left describe behavior attached to a *press*, *move*, and *release* event, just as an Elements, Events & Transitions model at the same level of abstraction would. We may adapt this language as an underlying editable graphical notation for Grizzly Bear in the future.

APPENDIX C

Application Code for Section III.3.9

The last paragraph of Section III.3.9 refers to application code that implements the enabling and disabling of two buttons. This appendix will list the actual programming language code (C++), which presents a minimal example of code integrated with an EET model, and the application programmer interface (“API”) for doing so.

```
void App::event(
      const Event& e,
      const ScValueContext& c,
      const ScBValueContexts& cs)
{
   if(e.getEvent()==”checkLinkButtons”)
   {
      NodesAndLinksAppPrivate::checkLinkButtons(e,c,cs);
   } else {
      cerr << “Warning: unknown event:\n “ << e << “\n”;
   }
}
```

The “App::event” method above is the entry point for all application code. It delivers the event thrown towards the application (“e”), as well as contextual information about the EET model which generated the event (“c”) and information about all known EET models (“cs”). Application code can then either change EET models directly through the “c” and “cs” pointers, or throw events back towards the EET interpreter (the latter method is use below).

```
void NodesAndLinksAppPrivate::checkLinkButtons(
      const Event&,
      const ScValueContext&,
      const ScBValueContexts& cs)
```

```
{
   const ScValueContext* ic = cs.findTag(“imodel”);
   if(!ic) {return;}
   const ScValueContext& c = *ic;

   const BElements& es = c.getElements();
   const BElements& aes = c.getAbstractElements();

   string fromNode;
   string toNode;
   if(thereAreExactlyTwoSelectedPins(aes,fromNode,toNode) &&
      thereIsALinkBetweenThePins(es,fromNode,toNode))
   {
      if(!theDeleteLinkButtonIsEnabled(es))
      {
         c.throwEvent(cs,new Event(“DeleteLinkButton”,”enable”));
      }
   } else {
      if(theDeleteLinkButtonIsEnabled(es))
      {
         c.throwEvent(cs,new Event(“DeleteLinkButton”,”disable”));
      }
   }

   if(thereAreExactlyTwoSelectedPins(aes,fromNode,toNode) &&
      !thereIsALinkBetweenThePins(es,fromNode,toNode))
   {
      if(!theNewLinkButtonIsEnabled(es))
      {
         c.throwEvent(cs,new Event(“NewLinkButton”,”enable”));
      }
   } else {
      if(theNewLinkButtonIsEnabled(es))
      {
         c.throwEvent(cs,new Event(“NewLinkButton”,”disable”));
      }
   }
}
```

The remaining procedures help implement the logic by accessing the state of the elements. They can access EET modelling constructs such as *element* and *attribute* by including the appropriate header files of the EET run-time library.

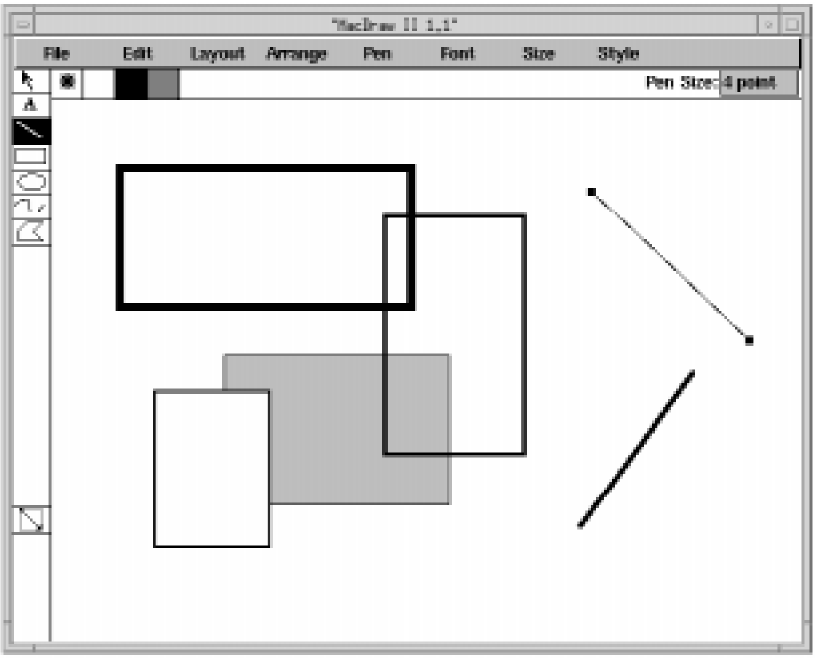
```
int thereAreExactlyTwoSelectedPins(
      const BElements& aes,
      string& fromNode,
      string& toNode)
```

```
{
   const Element* e = aes.lu(“bb”);
   if(!e) {return 0;}

   const Attribute* a = e->getAllowMultiple(“currentlySelectedNodes”);
   if(!a) {return 0;}

   const Strings& vs = a->gv();
   if(vs.length()!=2) {return 0;}

   fromNode = *vs.first();
   toNode = *vs.second();
   return 1;
}

int thereIsALinkBetweenThePins(
      const BElements& es,
      const string& fromNode,
      const string& toNode)
{
   ConstIterator<Element> i(es);
   const Element* e;
   while(e=i()) {
      const string& from = e->getv(“from”);
      if(from==””) {continue;}
      const string& to = e->getv(“to”);
      if(to==””) {continue;}

      if(!((from==fromNode && to==toNode) ||
           (from==toNode && to==fromNode))) {continue;}

      return 1;
   }
   return 0;
}

int theDeleteLinkButtonIsEnabled(
      const BElements& es)
{
   const Element* e = es.findElementByName(“DeleteLinkButton”);
   if(!e) {return 0;}
   const string& v = e->getv(“status”);
   return v==”enabled”;
}

int theNewLinkButtonIsEnabled(
      const BElements& es)
{
   const Element* e = es.findElementByName(“NewLinkButton”);
   if(!e) {return 0;}
   const string& v = e->getv(“status”);
   return v==”enabled”;
}
```

APPENDIX D

A “MacDraw” Imitation driven by an EET Model

We re-implemented MacDraw’s user interface on the basis of our Elements, Events & Transitions (EET) model to validate the expressiveness of the language.[90] Figure D-1 shows a snapshot of this interface.



Pen Size: 4 point

Figure D-1: A “MacDraw” Imitation

We implemented the following subset of MacDraw’s features. The vertical palette in the upper left-hand corner exactly imitates MacDraw’s equivalent. (A single click on e.g. the line item highlights it in gray, meaning that you can create

---

[90] We actually imitated the interface of MacDraw II 1.1.

exactly one new line. A double-click highlights it in black, meaning that you are in “line creation mode” until you explicitly leave it by clicking on another palette item.) The horizontal palette sets the fill color of the currently selected objects to none, white, black or gray. (MacDraw offers more choices, but they work in the same way.) Users can set the line width of the currently selected objects from the “Size” menu. Elements are created in the same way as in MacDraw (however, we limited ourselves to implementing rectangles and lines). By default, pressing a mouse down button sets one corner of the rectangle that is created. The size of the rectangle then changes continuously in response to mouse movement events until the mouse button is released (creating lines works similarly). Just as in MacDraw, clicking on the icon shown in the lower left-hand corner of Figure D-1 changes this behavior so that the first click defines the *center* of the object rather than a corner (clicking it again reverts to the default behavior). Clicking on an object selects it and deselects all others. The selection status is shown via handles. Multiple objects can be selected by shift-clicking on them (which toggles an object’s selection status without affecting others). Finally, objects can be moved (a “ghost frame” is shown while dragging just like in MacDraw), and the currently selected objects can be deleted by pressing the “Delete” key on the keyboard.

The EET model that drives this behavior consists of one abstract element and forty-five transitions (approximately five-hundred lines of EET code). We have built the model through a combination of demonstration and textual editing, partially because the demonstrational components were not complete at that point.

With one exception, all of this model can now be demonstrated to Grizzly Bear.[91] However, doing so without also being able to read the generated textual output is virtually impossible, so that we do not claim that constructing this complex EET-based application can be easily mastered by a beginning designer (it took us two days). It appears that combining Grizzly Bear's inferencing strength with a visual language is the most promising approach towards this goal.

---

[91] The exception is that the meaning of a "double-click" cannot currently be demonstrated because this would require a "simulated clock" mechanism for demonstrations (the *real* time that passed between two examples of a "double-click" demonstration is of no interest). Grizzly-Bear can of course already tie behavior to double-clicks if double-click events are built into the underlying toolkit (as done in Garnet [Myer90b], for example).

APPENDIX E

Material and Data from the Non-Programmer Study

E.1 Questionnaire

1. What is your major school? (circle one)

Psychology Other: ____________________

2. What is your year in school? (circle one)

Freshman Sophomore Junior Senior Grad Student

3. How old are you? (in years, e.g. 29)

____

4. Are you... (circle one)

female male

5. During your most intense week of computer use, how many hours per week did you use a computer? (circle one)

0 <10 <20 <30 <40 <50 <60 >=60

6. How many days has it been since you last used a computer?

____

7. Which of the programming languages below are you most familiar with? (circle only one)

a. Pascal *b*. Modula *c*. C/C++ *d*. LISP *e*. Basic *f*. Eiffel
g. Java *h*. TCL *i*. Assembler *j*. Prolog *k*. Cobol *l*. Forth
m. Other: ____________________
n. I am not familiar with any programming language.

8. During your most intense week of programming in this language, how many hours per week did you use this language? (use 0 if you chose "n." above)

0 <10 <20 <30 <40 <50 <60 >=60

9. How many days has it been since you last programmed in this language? (put down "not applicable" if you chose "n." above)

____

E.2 Experiment Instructions

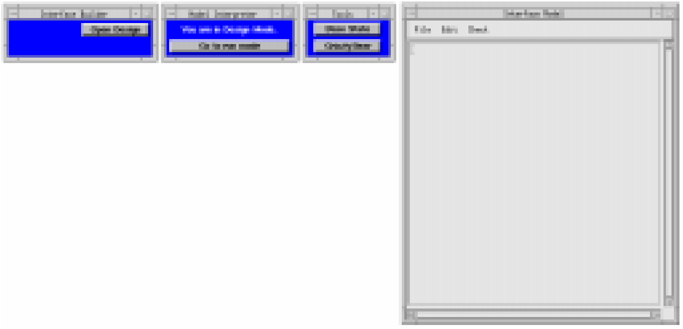
In this experiment, you will program small fragments of user interface behavior. The experiment consists of two phases. First, you will follow step-by-step instructions for three training tasks (Training Task A, B and C). You are than asked to accomplish four tasks on your own (Task 1, 2, 3 and 4).

Your objective should be to accomplish as many of the four "real" tasks as you can. The experiment ends after sixty minutes (or earlier if all four tasks are complete).

It is worth remembering that the experiment is designed so that it is quite difficult to complete all four tasks in an hour, and that the reason that the experimenters won't help you with the tasks is because that would distort the experiments (not because they are mean-spirited). Above all, it is worth remembering that we are evaluating the *system* - not you.

How to get to the Base State

What you now see on your screen should look like the picture below.



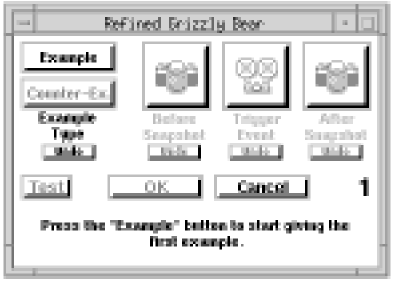
This is the state you should be in before you start with any new task. You can use the following check-list to get to it. (You don't need to do any of this now, or even read it, this is just for future reference.)

- **Click on the "Base State" button**. Everything disappears.
- Wait until a blue window appears, then **click its "More-->" button**.
- Wait until another blue window appears, then **click its "More-->" button**. You are back in the Base State.

Grizzly Bear - The Big Picture

Let us provide an overview of the tool you are going to use before we walk you through three examples.

Grizzly Bear is a tool that lets you “program” user interface behavior without having to use a textual programming language. This “programming” is done by giving examples of how you want the user interface to behave.



Press the "Example" button to start giving the first example.

Giving one example consists of going through the iconic buttons from left to right. In every example, you essentially say “*if* the interface is in this state (Before Snapshot) *and* the following event occurs (Trigger Event) *then* the interface should go to this state (After Snapshot)”.

After clicking any of the iconic buttons, you can always press the little “Undo” button under the currently-enabled button for un-doing the last step (say if you recorded an erroneous event or snapshot).

After giving an example, you can test if the user interface indeed behaves as you expected (by pressing the “Test” button and by then interacting with the user interface). If it doesn’t, you can typically achieve what you want by providing more examples.

If you give more than one example, you may want to provide different Before snapshots to be able to say “if the interface is in this state and I click here then ..., but if the interface is in *this* state and I click here then...”.

Finally, counter-examples are useful in defining behavior that applies to a whole class of objects. This is done by first giving examples of that behavior for some representative objects of this class as usual, and by then giving *counter-examples* for objects that do *not* exhibit the behavior. Grizzly Bear than guesses how to tell the objects in the examples from the objects in the counter-examples.

Training Task A

How to set up

Make sure you are in the Base State described on page 1. Click on the “Open Design” button. A file selection dialog box will pop up. Select the entry “TrainingA.sce” by clicking on it (it will highlight in black). Then click on the “OK” button. A new window labelled “Training Task A” will appear on your screen.

What your task is

Your task is to make pressing on the circle have the effect of changing its color to blue.

Step-by-step instructions (Training Tasks Only)

Make sure you have already opened the "Training Task A" window on your screen at this point by following the instructions under "How to set up" above.

Click on the "GrizzlyBear" button. A window labelled "Refined Grizzly Bear" appears.

Press on the "Example" button. It tells Grizzly Bear that you are about to give an example.

Press on the "Before Snapshot" button. We could change the user interface to set up for an example before pressing this button, but we don't need to here.

We will now demonstrate which event will trigger the behavior we are about to define (namely pressing on the circle). The next paragraph contains instructions that are time-critical, so we suggest you first read the next paragraph in its entirety to know what you will be doing, *then* go do it.

* The time-critical sequence is: **click on the recorder icon titled "Trigger Event"** (the interface starts beeping and the time-bar next to the recorder icon starts shrinking), then **press the left mouse button down on the red circle** *and hold it down without moving the mouse until the beeping stops*.

(end of time-critical seq.)

If the status line at the bottom of the "Refined Grizzly Bear" window now reads "What should happen when 'circle1' is 'pressed' ?" then please go on to the next paragraph.

Otherwise, if the status line reads "Hey! You did not cause an event!" you probably didn't act fast enough after clicking the "Trigger Event" icon - please go back to the paragraph marked with an asterisk (*) and try again.

Or, if the status line reads "What should happen when ... is ... ?" but not "when 'circle1' is 'pressed '?" then you recorded a different event - click the small "Undo" button under the "After Snapshot" label now, and then go back to the paragraph with the asterisk.

Click on the red circle to select it - several diagonal lines through it will tell you that it is highlighted (try clicking it again if it doesn't work the first time). Now **hold down the right mouse button over the circle and select "Properties" from the menu that appears**. A window labelled "Properties of SXCircle" appears. **Click on the "COLORS" tile** there (it then appears pushed in). Now **click on the "fillForeground" tile** (it should now also appear pushed in, and the textfield should now say “red”). **Move the mouse cursor right behind the color string** (“red”), **click there**, **repeatedly hit the “BackSpace” key** on the keyboard to delete the current color, **type in “blue”**, and **hit the Return key**. The circle’s color should now have changed to blue. (You can drag the properties window by its title bar to move it out of your way.)

Click on the icon titled “After Snapshot” to tell Grizzly Bear that you are done editing. After this example, Grizzly Bear resets the interface to where it was before the example (the circle appears red again), and displays its guess of which behavior you have in mind in the “Interface Model” window in textual form. You can now test the behavior interactively **by clicking on the “Test” button in the “Refined Grizzly Bear” window**. Please do so now (and wait until the status line reads “You can now interact with the design. ...”). Then **click on the circle**, and it should indeed turn blue as we have demonstrated. **Click on the “Back” button** to go back to demonstration mode (that’s the same button that said “Test” earlier).

Use the instructions on page 245 to reset to the Base State.

Training Task B

How to set up

Make sure you are in the Base State described on page 1. Click on the “Open Design” button. A file selection dialog box will pop up. Select the entry “TrainingB.sce” by clicking on it (it will highlight in black). Then click on the “OK” button. A new window labelled “Training Task B” will appear on your screen.

What your task is

Your task is to make the yellow circle move to wherever you press the left mouse button on the dark gray background of the “Training Task B” window.

Step-by-step instructions (Training Tasks Only)

Make sure you have already opened the “Training Task B” window on your screen at this point by following the instructions under “How to set up” above.

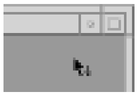
Click on the “GrizzlyBear” button to bring up the “Refined Grizzly Bear” window (drag it by its title bar if it is in your way). **Press the “Example” button**. **Press the “Before Snapshot” button**. As in the first training task, the sequence in the next paragraph is time-critical.

* **Press the “Trigger Event” button**, then **press the left mouse button over the upper-**

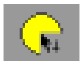
right hand corner of the “Training Task B” window and *hold the button down without moving the mouse until the beeping stops*.

(end of time-critical seq.)

If an icon appears there as shown in the picture below, please go on to the next paragraph. Otherwise, if the status line reads “Hey! You didn’t record an event!” please try again starting at the asterisk (*). If you accidentally recorded a different event, press the “Undo” button under the “After Snapshot” label and then re-try from the asterisk on.



Select the yellow circle with the left mouse button (it should appear “striped”, otherwise try again). Then **move the circle by dragging it with the middle mouse button** (position its center under the tip of the mouse cursor imprint as shown in the picture below).



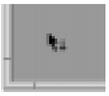
Click the “After Snapshot” button. Grizzly Bear will then reset the interface to the last Before snapshot (the circle appears at its original position), and show you the inferred program in the “Interface Model” window. Check out this inference now **by clicking on Grizzly Bear’s “Test” button** (then wait for two seconds). **Now click anywhere in the “Training Task B” window** (say near the center).

The problem is that Grizzly Bear thinks you meant that the circle always moves to the exact position you used in the example (rather than that it moves to where you click). The solution is to give it another example.

Click on the “Back” button to go back to demonstration mode, **click on the “Example” button**, and then **click on the “Before Snapshot” button**. We will now record another “press” event on the gray window, but we will use a different location this time. As usual, recording an event is time-critical.

* **Click on the Trigger Event button**, **then press the left mouse button on the gray window near its lower left-hand corner**, *and hold the mouse button down without moving the mouse until the beeping stops*.

The feedback on the press event should now look similar to the picture below, otherwise please record it again.



Now again move the center of the circle under the tip of the mouse cursor imprint **by first selecting the circle with the left mouse button** and **by then dragging it with the middle mouse button**. **Press the "After Snapshot" button** when you are done.

Grizzly Bear now displays its refined inference. Go to test mode **by pressing Grizzly Bear's "Test" button**. **Click at different places on the gray window** - the circle should now indeed follow the clicks. (If it appears a little bit off-center, that's ok.)

(Now reset to the Base State as usual, following the instructions on page 245.)

Training Task C

How to set up

Make sure you are in the Base State described on page 1. Click on the "Open Design" button. A file selection dialog box will pop up. Select the entry "TrainingC.sce" by clicking on it (it will highlight in black). Then click on the "OK" button. A new window labelled "Training Task C" will appear on your screen.

What your task is

The task is to make the circles change their color to blue when clicked.

Step-by-step instructions (Training Tasks Only)

Make sure you have already opened the "Training Task C" window on your screen at this point by following the instructions under "How to set up" above.

Bring up Grizzly Bear by pressing its button. **Press on the "Example" button**. **Press on the "Before Snapshot" button** (and wait for three seconds). **Record a "press" event on one of the five circles**. **Turn the circle you pressed blue** (click on the circle with the left mouse button [must appear striped, otherwise try again], select "Properties" from its right-button menu, press "COLORS" there, press "fillForeground", replace "beige" with "blue", and hit the Return key on the keyboard, then move the Properties window out of your way). **Press the "After Snapshot" icon**.

Go to test mode (click on the test button). **Click on some circles**. Grizzly Bear so far assumes that only the *particular* circle you used in the example can be turned blue. In order for it to generalize this behavior to *all* circles you have to also give it at least one counter-example - an example of an object which does *not* exhibit this behavior.

To do so, **go back to Demo Mode by pressing "Back"**, **press the "Counter-Example" button**, then **press the "Before Snapshot" button**, and then **record a "pressed" event on one of the rectangles**. Grizzly Bear then immediately shows its generalized inference (no After Snapshot is required for counter-examples). It should now have inferred the intended behavior - test it **by clicking "Test"** and **going through the "When you know you are done" procedure below**.

When you know you are done

In test mode, press on the five circles one by one, then click on the five rectangles one by one. You are done if all of the five circles turned blue when pressed while nothing happened when the rectangles were pressed.

(Finally, reset to the Base State following the instructions on page 245.)

Task 1

How to set up

Make sure you are in the Base State described on page 1. Click on the "Open Design" button. A file selection dialog box will pop up. Select the entry "_Task1.sce" by clicking on it (it will highlight in black). Then click on the "OK" button. A new window labelled "Task 1" will appear on your screen.

What your task is

Make pressing on the yellow rectangle change its color to green.

What you have to know

The color property to change is called "fillForeground". Make sure you are not changing a different color property by checking if the color you are replacing says "yellow".

When you know you are done

Go to test mode (click on the "Test" button in the "Refined Grizzly Bear" window). Then click on the yellow rectangle. You are done if its color indeed changed to green (check the box as a reminder that this task is complete).

☐

If you just can’t get it to work

If you’re not sure how to demonstrate to Grizzly Bear you may want to go through a training task again (the one most similar to this task, preferably).
It may also help to sit back and think about how to demonstrate the behavior before you actually start demonstrating it (the “Big Picture” section of page 245 may help).
Finally, if you just can’t get it to work at all, try another task and return to this one afterwards.

Task 2

How to set up

Make sure you are in the Base State described on page 1. Click on the “Open Design” button. A file selection dialog box will pop up. Select the entry “_Task2.sce” by clicking on it (it will highlight in black). Then click on the “OK” button. A new window labelled “Task 2” will appear on your screen.

What your task is

Your task is to program that clicking on any of the red circles will make them disappear - while making sure that nothing happens to the yellow circles when clicked.

What you have to know

In order to remove a circle, first select it with the left mouse button (it should appear “striped”, otherwise try again). Then hold down the right mouse button and select “Cut” from the menu that pops up.

When you know you are done

Go to test mode (click on the “Test” button in the “Refined Grizzly Bear” window). Click all the circles one by one. You are done if all of the red circles disappear when clicked while nothing happened to the yellow circles (check the box as a reminder that this task is complete).

☐

If you just can’t get it to work

If you’re not sure how to demonstrate to Grizzly Bear you may want to go through a training task again (the one most similar to this task, preferably).
It may also help to sit back and think about how to demonstrate the behavior before you actually start demonstrating it (the “Big Picture” section of page 245 may help).
Finally, if you just can’t get it to work at all, try another task and return to this one afterwards.

Task 3

How to set up

Make sure you are in the Base State described on page 1. Click on the “Open Design” button. A file selection dialog box will pop up. Select the entry “_Task3.sce” by clicking on it (it will highlight in black). Then click on the “OK” button. A new window labelled “Task 3” will appear on your screen.

What your task is

Your task is to make a press on the gray background of the “Task 3” window produce a new purple circle there.

What you have to know

You can make a copy of the circle by first selecting it with the left mouse button (it should appear striped), and by then choosing “Copy” from its right-mouse-button menu. Now select the gray background with the left mouse button (it appears striped), and choose “Paste” from its right-button menu. An outline of the new circle now follows the mouse cursor, and you can drop it by pressing the left mouse button.

When you know you are done

Go to test mode (click on the “Test” button in the “Refined Grizzly Bear” window). Click on the gray background window in four different locations. You are done if the four clicks each produced a new circle where you clicked (check the box as a reminder that this task is complete).

☐

If you just can’t get it to work

If you’re not sure how to demonstrate to Grizzly Bear you may want to go through a training task again (the one most similar to this task, preferably).
It may also help to sit back and think about how to demonstrate the behavior before you actually start demonstrating it (the “Big Picture” section of page 245 may help).
Finally, if you just can’t get it to work at all, try another task and return to this one afterwards.

Task 4

How to set up (***follow carefully - differs from the other tasks!***)

Make sure you are in the Base State described on page 1. Click on the “Open Design” button. A file selection dialog box will pop up. Select the entry “_Task4.sce” by clicking on it (it will highlight in black). Then click on the “OK” button. A new window labelled “Task 4” will appear on your screen.

Now press the left mouse button down on the “File” menu in the “Interface Model” window, and release it over the “Open” menu item. Select “Task4.im” in that dialog box, and click “OK”. Some text appears in the “Interface Model” window.

The set-up for this task differs from the others because the interface design already exhibits some behavior (that you have just pre-loaded). Check out the current behavior by pressing the “Go to run mode” button now (and wait until it says “Back to design mode”). Click several times on a few of the circles and observe how they cycle through colors. Finally, press the big “Pressing here makes all red circles green.” button. It doesn’t do anything yet - this is going to be your task.

Press the “Back to design mode” button to leave the test-drive mode (and wait until it says “Go to run mode”).

What your task is

Make the “Pressing here makes all red circles green.” button do what it says: make all circles that happen to be red change their color to green.

What you have to know

The color property to change is again called “fillForeground”. Note that you can select multiple circles at once by Control-clicking them (then you can change the colors of multiple circles at once).

The button sometimes “gets stuck” after you press on it - ignore that weirdness, it otherwise behaves as it should.

When you know you are done

Go to test mode (click on the “Test” button in the “Refined Grizzly Bear” window). Press on the circles so that two are yellow, two are red, and two are blue. Press the button. Now make two different circles red. Press the button. You are done if in both cases pressing the button made the two red buttons turn green while others were not affected (check the box as a reminder that this task is complete).

☐

E.3 Data Gathered

Table E-1 presents the raw data gathered from the non-programmer study. The first nine rows correspond to the nine items in the questionnaire of Section E.1. The remaining six rows state how many of the four tasks the subjects could accomplish, and how long it took them to work through the instructions and through the individual tasks. (Section VI.1 describes the experiment in more detail.)

Table E-1: Data Gathered from the Non-Programmer Study

Subject	1	2	3	4	5	6	7	8	9	10
1. Major[a]	Mgmt	ISyE	EE	Und.	CmpE	AE	Bio	ChE	CE	CS
2. Year In School	3	2	2	2	2	2	2	2	2	3
3. Age	21	19	18	19	19	19	20	19	19	21
4. Gender	m	m	f	f	m	m	m	f	m	m
5. Comp. Intensity	20	20	30	20	50	20	30	20	40	60
6. Comp. Last	1	0	0	4	1	3	0	1	0	0
7. Prog. Language	Pascal	Pascal	C/C++	Basic	C/C++	Fortran	Basic	none	none	Pascal
8. Prog. Intensity	10	20	10	10	50	10	20	0	0	20
9. Prog. Last	547.5	100	24.5	21	90	60	100	n/a	n/a	240
Tasks Completed	3	3	4	1	3	2	3	2	4	4
Instructions	22:23	20:22	18:00	53:06	17:47	29:29	19:42	27:02	20:01	25:35
Task 1	2:23	1:26	1:35	1:31	1:41	2:10	2:16	2:25	4:43	1:48
Task 2	5:18	5:18	4:08		2:45	14:24	4:10	8:47	6:20	3:15
Task 3	4:07	7:55	7:09		3:34		6:40		6:38	3:11
Task 4			16:47						9:53	19:32

a. Legend: AE=Aerospace Engineering, Bio=Biology, CE=Civil Engineering, ChE=Chemical Engineering, CmpE=Computer Engineering, CS=Computer Science, EE=Electrical Engineering, ISyE=Industrial & Systems Engineering, Mgmt=Management, Und.=Undecided

APPENDIX F

Material and Data from the Programmer Study

F.1 Pre-Experiment Questionnaire

1. Have you used Inference Bear before?

yes no

2. Have you read a paper describing Inference Bear?

yes no

3. Have you attended one of my presentations which covered Inference Bear?

yes no

4. Have you ever used the SX/Tools user interface builder?

yes no

5. What is your approximate age? (Circle one - or omit if you'd rather not disclose.)

<24 25-29 30-34 35-39 40-44 45-49 >50

6. In the past, how many hours per week did you spend programming during your most intense project? (Circle one.)

0 (never programmed) <10 hrs/week <20 hrs/week <30 hrs week <40 hrs/week beyond 40

7. How long has it been since you last wrote a piece of code? (Choose only one.)

1) approx. ___ days
2) approx. ___ weeks
3) approx. ___ months
4) approx. ___ years
5) (I've never written code.)

8. Which, if any, of the following user interface tools have you used before? (Circle all that apply.)

HyperCard Macromind Director Mac-based Interface Builder
SX/Tools Motif Interface Builder Garnet
Others (Please scribble the ones you can think of down.)

F.2 Post-Experiment Questionnaire

1. Did you have fun using this environment?

1 (hated it) 2 3 (neutral) 4 5 (loved it)

2. Do you think this environment would help you build graphical user interfaces faster *than the tools you are using right now?*

yes no

3. Do you think this environment would help you build better *graphical user interfaces than the tools you are using right now?*

yes no

F.3 Data Gathered

Table F-1 presents the data gathered from the programmer study. There were four groups of subjects which all performed the same series of tasks. The first group used Inference Bear, Expression Finder and textual editing to accomplish the tasks (“ib-ef-t”). The second group used Grizzly Bear and textual editing (“gb-t”). The third group had to demonstrate the behavior to Grizzly Bear, it was expressly forbidden to make textual changes (“gb”). A final group of subjects was asked to textually program the behavior in the Elements, Events & Transitions language (“t”).

The table contains the subjects’ answers to the pre- and post-experiment questionnaires of Section F.1 and F.2, as well as the completion time for each task. (Section VI.2 describes the experiment in more detail.)

Table F-1: Data Gathered from the Programmer Study

	ib-ef-t 1	ib-ef-t 2	ib-ef-t 3	gb-t 1	gb-t 2	gb-t 3	gb 1	gb 2	t 1	t 2
Pre 1	no	no	no	yes	no	no	yes	no	no	no
Pre 2	no	no	yes	yes	no	no	yes	no	yes	yes
Pre 3	yes	yes	yes	yes	no	no	yes	no	yes	yes
Pre 4	no	no	yes	yes	no	no	yes	no	yes	no
Pre 5	30-34	25-29	25-29	25-29	25-29	45-49	30-34	30-34	35-39	30-34
Pre 6	<10	>=40	>=40	<40	>=40	>=40	>=40	>=40	<30	>=40
Pre 7	60	2	14	5	2	7	5	0	90	0
Pre 8[a]	2	4	1	4	1	2	4	0	3	4
Task 1	4:01	2:30	3:06	5:55	9:50	[b]**6:52**	[c]**9:55**	5:19	2:40	11:21
Task 2	2:39	1:55	6:12	1:56	1:24	[b]**3:40**	4:47	5:07	1:22	1:56
Task 3	10:07	4:10	2:51	4:56	10:32	[b]**11:20**	3:40	2:58	12:14	11:16
Task 4	5:40	7:23	10:21	1:35	8:46	[b]**6:24**	3:50	5:35	5:16	7:34
Task 5a	[c]**14:00**	3:13	21:36	5:47	8:50	[b]**4:20**	[c]**6:11**	4:28	3:50	4:35
Task 5b	[d]**15:55**	9:35	7:11	5:50	14:51	[b]**18:54**	21:49	[e]19:52	9:22	9:55
Task 5c	9:56	10:01	4:35	6:51	15:29	[b]**13:15**	[f]**17:12**	12:54	5:12	5:31
Task 5d	[g]**13:20**	10:55	5:17	1:50	4:55	[b]**4:30**	[g]**7:16**	5:16	4:31	4:02
Post 1	[h]-	-	-	5	4	4	3	1	4	5
Post 2	-	-	-	yes	yes	no	yes	no	yes	no
Post 3	-	-	-	no	no	no	no	yes	no	yes

a. The entry presents how *many* user interface building tools the subject had previously used (it does not list all the tools in the interest of saving space).

b. This subject received hints for virtually all tasks based on our memory (unfortunately, we did not originally record the nature of the hints and lost the video trace of this particular session).

c. Hint that the task requires two separate demonstrations.

d. Hint that the task is more easily accomplished through two separate transitions.

e. Hint that the task requires a negative example (a counter-example).

f. Hint to watch Grizzly Bear's status messages. (The version of Grizzly Bear used for the testing did not yet highlight important messages, see the improvements of Section VI.1.)

g. Hint on how to record a menu selection event.

h. The post-experiment questionnaire was not given to this group of subjects.

REFERENCES

[Astr75] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–587, October 1975.

[Byrn94] Michael D. Byrne, Scott D. Wood, Piyawadee “Noi” Sukaviriya, James D. Foley, and David E. Kieras. Automating interface evaluation. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 232–237, (Boston, Massachusetts, April 24-28) 1994.

[Codd70] Edgar F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cyph91] 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, Louisiana, April 28-May 2) 1991.

[Cyph93] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Massachusetts, 1993.

[Cyph95] Allen Cypher and David Canfield Smith. KidSim: End user programming of simulations. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 27–34, (Denver, Colorado, May 7-11) 1995.

[Fish92] Gene L. Fisher, Dale E. Busse, and David A. 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, California, November 15-18) 1992.

[Fole89] James D. Foley, Won Chul Kim, Srdjan Kovacevic, and Kevin Murray. Defining user interfaces at a high level of abstraction. *IEEE Software*, 6(1):25–32, January 1989.

[Fran93] Martin R. Frank and James D. 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, Georgia, November 3-5) 1993.

[Hart90] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3):181–203, July 1990.

[Hill86] Ralph D. Hill. Supporting concurrency, communication and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[Huds88] Scott E. Hudson and Roger King. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[Huds93] Scott E. Hudson. A system for efficient and flexible one-way constraint evaluation in C++. Technical Report 93-15, Graphics, Visualization & Usability Center, Georgia Institute of Technology, Atlanta, GA 30332-0280, 1993.

[Kier88] David E. Kieras. Towards a practical GOMS model methodology for user interface design. In Martin Helander, editor, *Handbook of Human-Computer Interaction*, pages 135–157. Elsevier (North-Holland), Amsterdam, 1988.

[Kim90] Won Chul Kim and James D. Foley. DON: User interface presentation design assistant. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 10–20, (Snowbird, Utah, October 3-5) 1990.

[Kodr88] Yves Kodratoff. *Introduction to Machine Learning*. Morgan Kaufmann, San Mateo, California, 1988.

[Koos85] Donald J. Koosis. *Statistics*. John Wiley & Sons, New York, third edition, 1985.

[Kosb94] David S. Kosbie and Brad A. Myers. Extending programming by demonstration with hierarchical event histories. In *Fourth International East-West Conference on Human-Computer Interaction*, pages 147–157, (St. Petersburg, Russia, August 2-5) 1994.

[Kova92] Srdjan Kovacevic. *A Compositional Model of Human-Computer Interaction*. PhD thesis, Dept. of EE&CS, The George Washington University, 1992.

[Kueh92] Thomas Kuehme and Matthias Schneider-Hufschmidt. SX/Tools - An open design environment for adaptable multimedia user interfaces. *Computer Graphics Forum*, 11(3):93–105, September 1992.

[Kurl93] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):277–304, October 1993.

[Lint89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, February 1989.

[Luo93] Ping Luo, Pedro Szekely, and Robert Neches. Management of interface design in HUMANOID. In *Proceedings of INTERCHI, ACM Conference on Human Factors in Computing Systems*, pages 107–114, (Amsterdam, The Netherlands, April 24-29) 1993.

[MacG91] Robert M. MacGregor. Using a description classifier to enhance deductive inference. In *Proceedings of the Seventh IEEE Conference on AI Applications*, pages 141–147, (Miami, Florida, February) 1991.

[Maul89] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proceedings of Siggraph*, pages 127–136, (Boston, Massachusetts, July 31-August 4) 1989.

[Myer88] Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.

[Myer89] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating graphical interactive application objects by demonstration. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 95–104, (Williamsburg, Virginia, November 13-15) 1989.

[Myer90a] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[Myer90b] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[Myer93] Brad A. Myers, Richard G. McDaniel, and David S. 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, April 24-29) 1993.

[Nard93] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press, Cambridge, Massachusetts, 1993.

[Olse86] Dan R. Olsen, Jr. MIKE: The menu interaction kontrol environment. *ACM Transactions on Graphics*, 5(4):318–344, October 1986.

[Olse90] Dan R. Olsen, Jr. and Kirk Allan. Creating interactive techniques by symbolically solving geometric constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 102–107, (Snowbird, Utah, October 3-5) 1990.

[Paus91] Randy Pausch, Nathaniel R. Young II, and Robert DeLine. SUIT: The Pascal of user interface toolkits. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 117–125, (Hilton Head, South Carolina, November 11-13) 1991.

[Sadu96] Erica L. Sadun. *DJASA - An Interactive Graphical Notation*. PhD thesis, College of Computing, Georgia Institute of Technology, 1996. (in progress).

[Sche86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Sing89] Gurminder Singh and Mark Green. A high-level user interface management system. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 133–138, (Austin, Texas, April 30-May 4) 1989.

[Sing90] 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, Utah, October 3-5) 1990.

[Suka90] Piyawadee "Noi" Sukaviriya and James D. Foley. Coupling a user interface framework with automatic generation of context-sensitive animated help. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 152–166, (Snowbird, Utah, October 3-5) 1990.

[Suka93] Piyawadee "Noi" Sukaviriya, James D. 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, April 24-29) 1993.

[Szek92] Pedro Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 507–515, (Monterey, California, May 3-7) 1992.

[Szek93] Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Proceedings of INTERCHI, ACM Conference on Human Factors in Computing Systems*, pages 383–390, (Amsterdam, The Netherlands, April 24-29) 1993.

[Szek95] Pedro Szekely, Piyawadee "Noi" Sukaviriya, Pablo Castells, Jayakumar Muthukumarasamy, and Ewald Sacher. Declarative interface models for user interface construction tools. In *IFIP Working Conference on Engineering for Human-Computer Interaction*, (Grand Targhee Resort, Wyoming, August 14-18) 1995.

[Ullm83] Jeffrey D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, second edition, 1983.

[Wiec89] Charles Wiecha, William Bennett, Stephen Boies, and John Gould. Generating highly interactive user interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 277–282, (Austin, Texas, April 30-May 4) 1989.

[Wiec90] Charles Wiecha and Stephen Boies. Generating user interfaces: Principles and use of ITS style rules. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 21–30, (Snowbird, Utah, October 3-5) 1990.

[Wolb91] 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, South Carolina, November 11-13) 1991.

VITA

Martin Frank was born in Mannheim, Germany, on August 4, 1966. He received his Abitur from the Carl-Benz Gymnasium Ladenburg in 1985, his Vor-Diplom in Informatik from the Universität Karlsruhe in 1988, and his Master's degree in Computer Science from the Georgia Institute of Technology in 1991.