A Survey of Three Dialogue Models

MARK GREEN
University of Alberta

A dialogue model is an abstract model that is used to describe the structure of the dialogue between a user and an interactive computer system. Dialogue models form the basis of the notations that are used in user interface management systems (UIMS). In this paper three classes of dialogue models are investigated. These classes are transition networks, grammars, and events. Formal definitions of all three models are presented, along with algorithms for converting the notations into an executable form. It is shown that the event model has the greatest descriptive power. Efficient algorithms for converting from the transition diagram and grammar models to the event model are presented. The implications of these results for the design and implementation of UIMSs are also discussed.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*languages*

General Terms: Algorithms, Design, Human Factors, Theory

Additional Key Words and Phrases: Dialogue models, human–computer interaction, user interface management

1. INTRODUCTION

Every user interface management system employs some model of user interfaces. This user interface model forms the basis of the notations used by the *user interface management system* (UIMS) for describing user interfaces and strongly influences its implementation. Both of these issues are worthy of further investigation. The notion of user interface models is developed in Green [18].

A number of techniques have been developed for describing user interfaces. These techniques can be divided into two broad classes, depending upon whether they are used in the design of user interfaces or in their implementation. Design notations can be very informal, since their main purpose is to record the thoughts of the designer. On the other hand, the notations used in the implementation of user interfaces must be formal, since they will be used to directly produce the implementation of the user interface. In this paper we are only concerned with

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

Author's Address: Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1.

the notations used in the implementation of user interfaces (this does not mean that design notations should not be considered by UIMS researchers).

The notations employed in a UIMS define the range of user interfaces that can be produced by that UIMS. In order to have a general UIMS (this may not always be an important goal), the design notations must be capable of describing the widest possible range of user interfaces. The range of a design notation can be measured in two ways. The first measure of range is the descriptive power of the notation. The descriptive power of a notation is the set of user interfaces that can be described by the notation. The larger this set is, the more powerful the notation. Determining the descriptive power of a design notation can be converted into a problem in formal language theory, and in most cases a definitive answer can be found. The second way of measuring the range of a design notation is by its usable power. The usable power of a design notation is the set of user interfaces that can *easily* be described by the design notation. The usable power of a design notation will always be a proper subset of its descriptive power. The design notation often exerts a subtle bias on the user interface designer. Most designers tend to favor user interfaces that are easy to describe over those that are hard to describe. Thus, if the best user interface for a particular application is very hard to describe in a given notation, there is a good chance that the user interface designer will not consider it. Unlike descriptive power, there is no objective way of measuring the usable power of a user interface. For this reason, descriptive power is used in this paper, with the possibility that some of the results could be misleading.

The implementation of a UIMS is often strongly influenced by the user interface model employed. The structure of the UIMS and of the services it provides often follows the structure of the user interface model. For example, if the user interface model is divided into a number of components, the UIMS will often supply design and implementation tools for each of these components. If each of the model components has a distinct function with well-defined interfaces, it is easy to construct a set of functionally equivalent tools for each component. This gives the designers some freedom in the choice of the tools that they use. If the designer is presented with a well-defined user interface model that seems logical, then he or she should have less trouble learning how to use the tools provided by the UIMS. This suggests that the design of a UIMS should start with the design of the underlying user interface model.

One of the best known user interface models is the Seeheim model [34]. This model is used as the basis for this paper for the following two reasons. First, the Seeheim model is fairly general and thus adequately describes a number of existing UIMSs. Second, it is one of the few user interface models with an explicit description that is independent of a particular UIMS. This facilitates the study of the model itself, without being influenced by its implementation in a particular UIMS.

The Seeheim model divides a user interface into three components, as shown in Figure 1. The presentation component deals with the physical representation of the user interface. This includes input and output devices, screen layout, interaction techniques, and display techniques. The presentation component is the only part of the user interface that deals directly with devices. The other components of the user interface cannot directly communicate with the input

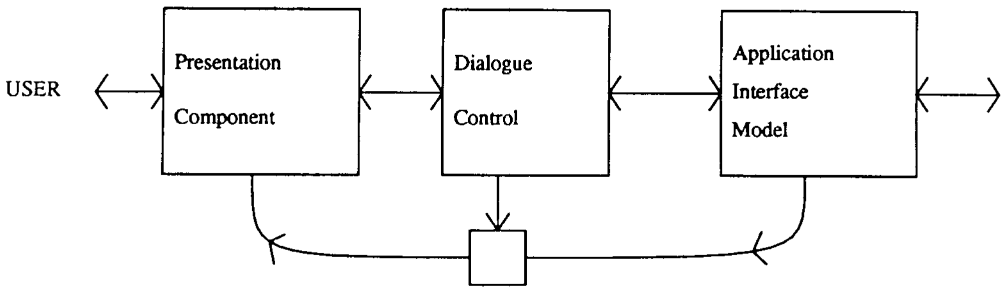


Fig. 1. The Seeheim model of user interfaces.

and output devices. The presentation component can be viewed as the lexical level of the user interface. The dialogue control component deals with the dialogue between the user and the computer system. This component is responsible for the structure of the commands and dialogue used by the user. It can be viewed as the syntactic level of the user interface. The application interface model defines the interface between the user interface and the rest of the program. This component handles the invocation of the application procedures. The three components can be logically viewed as three separate processes. The components communicate by passing tokens, which are similar to the tokens used in compilers [3]. A token consists of a name and a collection of data values. The name of the token identifies the type of the token. A token flowing from the user to the application program is called an input token, and a token flowing from the application program to the user is called an output token.

In the Seeheim model the application can change the state of the user interface by sending an output token to it. The user interface can view the application as another input device and react to it in the same way as it would to a human user. This use of output tokens provides a controlled mechanism for transferring state information from the application to the user interface. The user interface does not need to call routines in the application in order to determine its current state.

The main emphasis of most UIMSs has been on the dialogue control component. As a result, we know much more about this component than about the other components. The main emphasis of this paper is on the dialogue control component of user interfaces and the models that have been proposed for this component, which are called dialogue models. The three main dialogue models (transition networks, context-free grammars, and events) are defined in the next section. With the existence of three models we now have the problem of choosing the best model to use in a UIMS. What should be the basis of this choice? Previously, we noted that the notations employed in a UIMS must be as general as possible. So one criterion is the descriptive power of the model. If we can show that one model is more powerful than the other two, that model becomes the top candidate. If efficient procedures for converting descriptions in the other models to the more powerful one can be produced, the UIMS implementor only needs to implement the more powerful model, plus the conversion procedures. As a result, a UIMS can be constructed, based on the most powerful model, that will be capable of handling descriptions produced in all three models. In this paper we present formal definitions of all three dialogue models and some of their important properties. The major result of this work is that the event model is the most powerful, and that transition network and context-free grammar notations can be efficiently converted to the event notation.

2. DIALOGUE MODELS

The three types of dialogue models that are of interest to this work are transition networks, context-free grammars, and events. Informal descriptions of these three models are presented in this section, and formal definitions are presented in the next section.

2.1 Transition Networks

The *transition network* model is based on transition diagrams. A transition diagram consists of a set of states (represented by circles) and a set of arcs (represented by arrows leading from one state to another). The states represent the states in the dialogue between the user and the computer system. In the simplest form of transition networks, each arc is labeled by an action (an input token) that the user can perform. The arcs in the diagram determine how the dialogue moves from one state to another. The dialogue will move from state A to state B if there is an arc between these two states labeled by the action the user performed. A path through a transition diagram is a sequence of arcs that lead from the start state of the diagram to one of its final states. A sequence of user actions is accepted if they label the arcs on a path through the diagram.

A simple example of a dialogue described by a transition diagram is shown in Figure 2. The example used to illustrate all three dialogue models is entering a rubber band line. The presentation component produces the “button” token when the user presses the button on the locator and the “move” token each time the user moves the locator.

This simple form of transition network describes the sequences of actions that the user can perform but says nothing about the responses generated by the computer. One way of describing the computer’s side of the dialogue is to attach actions to the states in the diagram. When a state is reached, its action is executed. Thus, user actions are attached to arcs, and program actions are attached to states. By augmenting the previous example with program actions we have the diagram shown in Figure 3.

Program actions can also be attached to the arcs. In this case the program action is executed when the arc is traversed. Some UIMSs allow program actions to be attached to both arcs and states. In some cases attaching actions to arcs can result in smaller transition diagrams, which could be easier to understand. By allowing actions on arcs, the transition diagram for the example user interface can be reduced to three states (this is shown in Figure 4). In the following discussion we assume that the program actions are attached to the states. This simplifies the presentation of the algorithms. The algorithms presented in the following sections can easily be extended to cover the cases in which the actions are attached to the arcs (the algorithms were tested on transition diagrams with actions attached to the arcs).

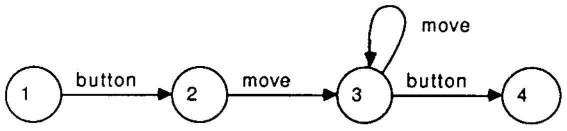
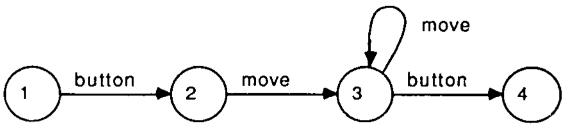
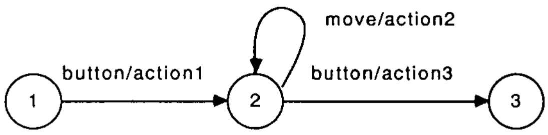


Fig. 2. A simple transition diagram.



2: record first point
3: draw line to current position
4: record second point

Fig. 3. Example transition diagram with program actions.



action1: record first point
action2: draw line to current position
action3: record second point

Fig. 4. Example transition diagram with actions on arcs.

There are two problems with this type of transition network. It can only describe a limited range of dialogues, and the descriptions tend to get quite large. The second problem can be solved by partitioning the network. One way of doing this is to divide the transition network into a number of diagrams. This results in a main diagram, plus a number of subdiagrams. A subdiagram is a complete transition diagram that can be invoked from another diagram. These subdiagrams can be viewed as procedures or subprograms. Any arc in a diagram can now be replaced by a call to one of the subdiagrams. An arc labeled by a subdiagram name will be traversed, when the user enters a sequence of actions that causes the subdiagram to be traversed, from its start state to one of its final states. An extended definition of arc traversal that covers subdiagrams can now be produced. An arc between two states will be traversed if its label is a user action and the user has performed that action, or if the arc's label is a subdiagram name and the user performs actions that cause the subdiagram to be traversed.

Subdiagrams make it possible to divide the description of the dialogue into a number of logical units. For example, there can be a subdiagram for each of the commands in a user interface. There can also be subdiagrams for common sequences of user actions (such as entering operand values). The use of subdiagrams does not increase the descriptive power of transition diagrams, since each call to a subdiagram can be replaced by the states and arcs in the subdiagram. If we allow subdiagrams to call themselves recursively, then the descriptive power is increased. A transition network notation that allows recursive calls is called a *recursive transition network* (RTN).

An extension of the rubber band example can be used to illustrate the difference between transition diagrams and recursive transition diagrams. In this example a sequence of rubber band lines is used to enter a polyline. At any point in the interaction the user can press the backspace key to remove the last point entered. The backspace key cannot be used to remove the first point in the polyline. The set of recursive transition diagrams for this example is shown in Figure 5. This dialogue cannot be described by transition diagrams, since with transition diagrams there is no way of telling when the user has backspaced to the last point of the polyline.

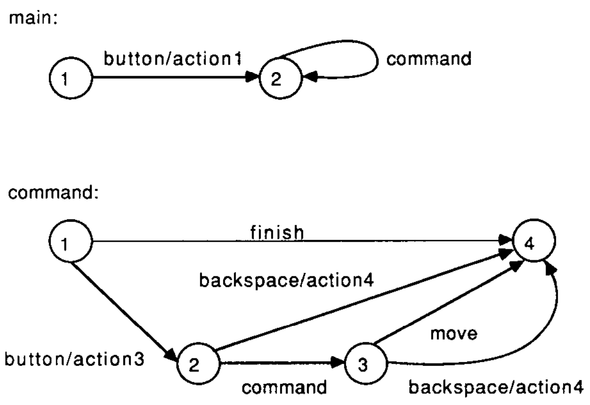
Augmented transition networks (ATN) are an extension of recursive transition networks [41]. An augmented transition network consists of a set of transition diagrams, a set of registers, and a set of functions. The registers can hold arbitrary values and are only visible within the dialogue control component (the application cannot read or write these registers). The functions can perform any computation on the register values, and they can assign new values to the registers. The functions cannot access data values in the application. Functions are attached to the arcs in the diagrams and are executed when the arc is traversed. If the value of the function attached to an arc is true, the arc can be traversed; otherwise, the arc cannot be traversed.

It is possible to construct context-sensitive dialogues with augmented transition networks (this is not the case for other types of transition networks). For example, in a database query application, the register values can indicate whether particular databases are open and the types of access privileges (read, write, control, etc.) that the user has. The register values can then be used to determine the commands that can be applied to each of the databases. Error messages can also be customized to take into account the access privileges of the user.

Augmented transition networks are more powerful than other types of transition networks. It has been suggested that they could be used as the basis for the dialogue control component of a UIMS (e.g., see Kamran [24], Kieras and Polson [27], or Sibert [36]). Jacob [23] has used a notation similar to augmented transition networks for specifying and prototyping user interfaces. The interactive pushdown automata suggested by Olsen [31] also resemble augmented transition networks.

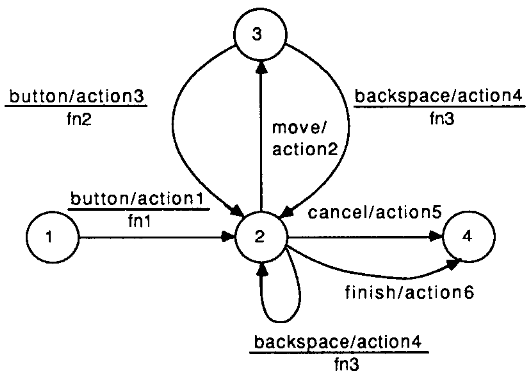
Augmented transition networks can easily handle the extended rubber band line example without the use of subdiagrams. A register can record the number of points that have been entered, and a function can check whether the register is greater than 1 when a backspace is entered. A further extension of this example is the addition of a cancel feature. Whenever the user presses the cancel button, the dialogue terminates with an empty polyline. This addition to the example cannot be handled by recursive transition networks owing to the nesting of the subdiagram calls. An augmented transition network for the cancel example is shown in Figure 6.

One of the major problems with the transition network model is handling unexpected user actions. When the diagram is at a particular state and the user action does not correspond to any of the arc labels, the dialogue cannot proceed.



action1: record first point
action2: draw line to current position
action3: record next point
action4: erase last point

Fig. 5. Transition diagram for polyline example.



action1: record first point
action2: draw line to current position
action3: record next point
action4: erase last point
action5: erase polyline
action6: return polyline

```
fn1: count:=1; return(true);
fn2: count:=count+1; return(true);
fn3: if count = 1 then
        return(false);
     else
        count := count-1;
        return(true);
```

Fig. 6. Polyline dialogue with cancel.

There are a number of solutions to this problem. The easiest solution is to ignore the user action. This is not very satisfactory, since the user is not informed that the program is ignoring his or her actions and has no idea of how to get to the next state. Another solution is to use a wild card label. This arc label will match any user action that does not match any of the other user actions. A wild card arc leads to a state where error recovery is attempted. Another approach to handling unexpected user actions are Olsen's pervasive states and transitions [31]. In this approach a transition diagram can have associated with it another transition diagram, which is used when the current input token does not match any of the arcs leaving the current state.

One of the main advantages of the transition network model is its natural graphical representation. Transition diagrams can easily be displayed and edited on graphics terminals. This advantage has (surprisingly) not been incorporated into a number of existing transition network based UIMSs.

The use of transition networks in user interfaces has a very long history. The first systems that could be called UIMSs were based on transition networks. The first use of transition networks in user interface design and implementation is the work of Newman [29, 30] and Parnas [33]. Since then, transition networks have been the basis for a large number of UIMSs (e.g., Kamran and Feldman [25] and Schulert et al. [35]).

The first use of recursive transition networks in user interfaces appears to be the work of Denert [10]. The SYNICS system is another early example of the use of recursive transition networks [9, 11, 20]. Experience with the use of both transition networks and recursive transition networks is presented in Kamran [24] and Sibert et al. [36].

2.2 Context-Free Grammars

The *context-free grammar* model is based on describing the dialogue between the user and the computer system by a context-free grammar. The basic motivation for this model is the view that human–computer interaction is a dialogue, as in human–human communication. In the case of natural languages a grammar is used to describe the language used by the participants in the dialogue. The natural extension of this idea is to use a grammar to describe the dialogue between the user and the computer. There is a major difference between human-computer interaction and human–human interaction. In the former case two distinct languages are involved, whereas in the latter only one language is used. That is, in human–computer interaction the user employs one language to enter commands to the program, whereas the computer uses another to communicate the results of these commands. The existence of two distinct languages causes numerous problems when grammars are used to describe user interfaces.

In practice, a grammar is used to describe the language employed by the user to communicate with the computer; the other direction is described by some other means. The terminals in the grammar are the input tokens generated by the presentation component. These tokens represent the user's actions. The terminals are combined by the productions in the grammar to form higher level structures called nonterminals. The collection of productions in the grammar define the language employed by the user in his or her interactions with the computer. An example of this type of description is shown in Figure 7. This example is the same as the one used for transition networks.

The grammar in Figure 7 describes the actions performed by the user in order to enter a rubber band line, but it does not cover the responses generated by the computer. In the case of transition diagrams, program actions are attached to the states in the diagram to indicate the program's responses. A similar approach can be used with context-free grammars; program actions can be attached to each of the productions in the grammar. These program actions are performed when the production is used in the parse of the user's input. One of the major problems with this approach is that the time when the production is used in a parse depends upon the parsing algorithm used. In the case of a bottom-up parse, a production is used when all the symbols on its right side have been recognized. In the case of a top-down parse, the production is used when the first terminal that could be generated by the right side is encountered. This issue is explored further in Section 3.2. By assuming that a top-down parse is used, the example in Figure 7 can be augmented by program actions to obtain the grammar shown in Figure 8. The grammar has been modified in order to produce the program actions at the appropriate place in the dialogue. This modification introduces three new nonterminals that do not contribute to the language but serve as place holders for the program actions. Most of the grammar-based UIMSs provide some notational mechanism for invoking actions in the middle of productions (see Olsen and Dempsey [32] for one possible mechanism). In these notations the introduction of new nonterminals for program actions is not necessary.

It is fairly easy to construct a context-free grammar for the polyline example presented in the previous section. The cancel extension to the polyline example cannot be described by a context-free grammar.

A number of extensions to context-free grammars have been proposed and implemented in some UIMSs [32, 39]. Most of these extensions deal with error recovery, undo processing, and control over the order of the parse. It should be noted that existing UIMSs do not use the full power of context-free grammars. The grammars used in these systems are restricted to the subset of context-free grammars that can be handled by the parsing techniques that they use (usually LL(1) or LALR(1)).

The first use of context-free grammars in the design and implementation of user interfaces is hard to determine. In the mid-1970s a number of research groups used commonly available parser generators, such as YACC [1], to produce the user interfaces to graphics programs. In most cases these experiments were a failure, owing to poor error recovery. The error recovery routines provided with these parser generators were designed for use with programming languages. When an error occurred, the recovery routine would ignore input tokens until a token in a recovery set (e.g., ";", "begin", and "end", in the case of programming languages) was encountered. While recovery was in progress, the user would receive no feedback and would often not know how to get back to an acceptable state.

An early example of a grammar-based system designed for user interfaces is one of the versions of the SYNICS system [12, 20]. Another early example of the use of context-free grammars is the work of Hanau and Lenorovitz [21].

```
line → button end_point

end_point → move end_point
          | button
```

Fig. 7. Context-free grammar for rubber band line example.

```
line → button d1 end_point

end_point → move d2 end_point
          | button d3

d1 →
     { record first point }

d2 →
     { draw line to current position }

d3 →
     { record second point }
```

Fig. 8. Rubber band line example with program actions.

The SYNGRAPH system developed by Olsen has a number of interesting properties [32]. The grammar is used to produce a top-down parser for the dialogue. The designer of the dialogue can specify the error recovery mechanism used by the parser, allowing for more natural and graceful error recovery. The grammar is also used to produce some of the information required by the presentation component. This information includes the contents of menus and when devices must be acquired and released.

Input–output tools [39, 40] and *dialogue cells* [6, 38] take a different approach to the specification of a grammar-based dialogue. In this approach each production in the grammar is enclosed in an input–output tool or dialogue cell. The information accompanying a production specifies the actions (prompting and initialization of local variables) performed when the production becomes active (able to take part in the parse), the action performed when the end of the production is reached, the echo produced for the interaction, and the value returned by the production. A top-down parser is constructed from the grammar.

2.3 Events

The *event* model is not so well established as the other two dialogue models. This model is based on the concept of input events that is found in a number of graphics packages. In these packages the input devices are viewed as sources of events. Each input device generates one or more events when the user interacts with it. An event has a name or number that identifies the nature of the interaction, plus several data values that characterize the interaction. For example, in the case of a tablet, a move event is generated each time the tablet cursor is moved. The data associated with this event are the x and y coordinates of the cursor. A separate event is also generated each time one of the cursor buttons is pressed or released.

The events are placed on a queue when they are generated, and the application program removes the events one at a time from the queue by calling one of the routines in the graphics package. In some cases the application program can specify the type of event it requires, giving the application program some control over the events that are generated. In these graphics packages there are a fixed number of predefined events, and they can only be generated by input devices.

The event model is an extension of this basic idea. In the event model there is an arbitrary number of event types. Some of the events are generated by input devices, and other events are generated inside of the dialogue control component. The programmer is free to define new event types that are more appropriate for a particular application. In the event model there are no explicit queues of events. When an event is generated, it is sent to one or more *event handlers*. An event handler is a process (defined by a procedure or module) that is capable of processing certain types of events. When an event handler receives one of the events it can process, it executes a procedure (this procedure is similar to a method in Smalltalk [15]). This procedure can perform some computation, generate new events, call application procedures, create new event handlers, or destroy existing event handlers.

The behavior of an event handler is defined by a template. A template consists of several sections that define the parameters to the event handler, its local variables, the events it can process, and the procedures used to process these events. When an event handler is created, its template must be specified, along with values for its parameters. The result of the creation process is a unique name that is used to reference the event handler. Several event handlers can be created from the same template. Each of the event handlers created from a template can have a different local state (parameter and variable values).

Once an event handler has been created, it is in the active state. It remains in this state until it is destroyed, either by itself or by another event handler. Only the active event handlers can respond to events. In the event model a user interface is described by the set of templates that define the event handlers it uses. At the start of execution an instance of one of these templates is created to serve as the main event handler in the user interface. This event handler will then create (possibly indirectly) all the other event handlers in the user interface. Conceptually, all the event handlers in the user interface execute concurrently, processing the events as they arrive. An event handler can only process one event at a time; so it can be viewed as a monitor.

There are two possible sources of events in the dialogue control component. The first source is the other two components of the user interface. The presentation component and the application interface models both send tokens to the dialogue control component. These tokens are converted into events, which are then sent to the event handlers that process them. An event handler can declare the tokens that it is interested in. When one of these tokens arrives, it is converted into an event and processed by the event handler. More than one event handler can process the same token. The second source of events is from within the dialogue control component itself. An event handler can send an event to one of the other event handlers in the dialogue control component. In this case the name of the event handler that receives the event is explicitly specified in the event send operation. This form of event is used for communications between the event handlers.

An event handler for the rubber band line example is shown in Figure 9. A notation similar to that presented in Green [19] is used in this example. The

```
EVENT HANDLER line;

 TOKEN
  button Button;
  move Move;

 VAR
  int state;
  point first, last;

 EVENT Button  DO {
  IF state == 0 THEN
   first = current position;
   state = 1;
  ELSE
   last = current position;
   deactivate(self);
  ENDIF;
 };

 EVENT Move DO {
  IF state == 1 THEN
   draw line from first to current position;
  ENDIF;
 };

 INIT
  state = 0;

END EVENT HANDLER line;
```

Fig. 9. Event handler for the rubber band line example.

```
EVENT HANDLER polyline_cancel;

 TOKEN
  button Button;
  move Move;
  backspace Backspace;
  cancel Cancel;
  finish Finish;

 VAR
  point_count : integer;
  point_list : list of point;
  int state;

 EVENT Button DO {
  IF state = 0 THEN
   point_list = current position;
   state = 1;
   point_count = 1;
  ELSE
   add current position to point_list;
   point_count = point_count + 1;
  ENDIF;
 };

 EVENT Move DO {
  IF state = 1 THEN
   draw line from last position to current position;
  ENDIF;
 };

 EVENT Finish DO {
  return(point_list);
  deactivate(self);
 };

 EVENT Backspace DO {
  IF point_count > 1 THEN
   remove last point from point_list;
   point_count = point_count - 1;
  ELSE
   output "can't delete first point";
  ENDIF;
 };

 EVENT cancel DO {
  return(empty_list);
  deactivate(self);
 };

 INIT
  state = 0;

END EVENT HANDLER polyline_cancel;
```

Fig. 10. Event handler for polyline with cancel.

definition of the line event handler is divided into four sections. The first section declares the tokens that the event handler can process. This section maps the tokens "button" and "move" into the events Button and Move. The VAR section consists of the declarations of the event handler's local variables. The third section of the declaration specifies the processing to be performed on the events received by the event handler. The last section contains the statements to be executed when an event handler is created. Note that in this example the program responses are included with the processing of the user's input.

An event handler can be constructed for the polyline example with the cancel extension. This event handler is shown in Figure 10.

One of the main advantages of the event model is its ability to describe *multithreaded dialogues*. In a multithreaded dialogue the user can be involved in several separate or communicating dialogues at the same time. The user is free to switch from one dialogue to another at any point in the interaction. This type of dialogue occurs frequently in window-based systems. Since each of the event handlers can be viewed as a separate process, it is quite easy to describe multithreaded dialogues in the event model.

As an example of a multithreaded dialogue consider a text editor that allows the user to edit several files simultaneously. Each of the files is displayed in a separate window. It is possible to produce a collection of event-handler templates that describe the interactions that occur in an editing window. When the user edits a file, instances of these event handlers are created. Thus each editing window has its own collection of event handlers. Editing within a single window can be described in any of the three models, but the transition network and grammar models have trouble with communications between the windows. For example, consider a cut-and-paste feature, which could be part of the multifile editor. The user can cut or copy some text from one file to another. This involves communications between the dialogues running in two windows. In the event model this can easily be handled by sending events from one window to another, but it cannot be described in the other two models.

The event model was motivated by early work on Smalltalk [26]. One of the first implementations of this model was the work of Green [16]. An elaboration of this version of the event model was used in the University of Alberta UIMS [19] and is formalized in Section 3.3 of this paper. Two other UIMSs that support the event model are Sassafras [22] and ALGEA [14]. An early use of events and event handlers in the construction of user interfaces is described in [13].

A similar model was developed by Anson. Anson [4] has used events to define the semantics of graphical input devices. He has also used object-oriented languages to extend this model to the description of dialogues [5]. Cardelli and Pike [7] have produced a language called Squeak for processing the input from mice and keyboards. This language is based on processes and messages between processes. Their processes are similar to event handlers, and messages serve the same purpose as events. Tanner et al. [37] have also described a multiple-process and message-based system to support graphics and user interfaces. This system is based on the Harmony operating system. There is a difference between the messages used in these systems and events. In Squeak a message will not be sent if there is not another process ready to receive it. In Harmony the sending process blocks until the receiving process receives the message and replies. An event is sent regardless of whether the receiver is ready for it, and the event send cannot block. It is not clear whether the difference between messages and events is important.

3. FORMAL DEFINITIONS OF THE DIALOGUE MODELS

In this section formal definitions of the three dialogue models are presented. The main reasons for producing these definitions is to provide both a formal framework for comparing the three models and a means of deriving their properties. The main emphasis in these definitions is on the mechanism used to describe the actions performed by the user. In these definitions it is assumed that the applications program cannot directly change the state of the user interface. In order to change the state of the user interface the application must send an output token to the user interface. The main implication of this assumption is that the user interface cannot use application procedures to extend the power of the user interface model. If this type of behavior is allowed, then the notion of a separate user interface module breaks down.

3.1 Transition Network Model

The development of the formal definition of the transition network model starts with simple transition networks (transition networks with only one diagram) and is extended to include recursive transition networks.

In the case of simple transition networks their behavior can be described by a *modified finite-state machine*. In this finite-state machine the states correspond to the states in the transition diagram, and the arcs are converted into transitions between these states. The input alphabet is the set of input tokens that can be generated by the presentation component. On the basis of this representation a *simple transition network* (STN) is a 7-tuple $M = (Q, \Sigma, P, \delta, \gamma, q_0, f)$, where

(1) Q is a finite set of states corresponding to the states in the diagram;
(2) Σ is a finite set of input symbols, which are the input tokens generated by the presentation component;
(3) P is a finite set of actions, which are the actions labeling the states of the transition diagram;
(4) δ is a mapping from $Q \times \Sigma$ to Q, called the state transition function;
(5) γ is a mapping from Q to P, called the action function;
(6) $q_0 \in Q$ is the initial state of M;
(7) $f \subset Q$ is the set of final states of M.

The behavior of a STN is slightly different from that of a finite-state machine. The configuration of a STN is represented by the current state q of the machine. When the STN receives an input symbol a, representing an action performed by the user, it changes to the state given by the state transition function. In making this state transition, the STN emits the name of the action labeling the new state. That is, the value of $\delta(q, a)$ is the new state q', and the action p emitted is given by $\gamma(q')$.

At the start of the dialogue the STN is in the initial state q_0. As the user generates input tokens, the STN will move from one state to another, following arcs labeled by the input tokens. The end of the dialogue occurs when the STN enters one of the final states in the set f. The STN recognizes the strings of input tokens that cause it to move from its initial state to one of its final states.

In the case of recursive transition networks a more powerful mechanism is required. This mechanism is the deterministic push-down automaton (DPDA) [2]. A modified version of the DPDA, called a transition network (TN), is used in our definition of recursive transition networks. The main difference between a STN and a TN is the use of subdiagrams in TNs. In a recursive transition network, when an arc labeled by a subdiagram name is encountered, control is transferred to the subdiagram. When the traversal of the subdiagram is complete, control transfers back to the node at the end of the arc invoking the subdiagram. Since subdiagrams can be invoked recursively, some mechanism must be provided for storing the nodes at the end of the invoking arcs. This is the main reason for using a DPDA as the basis of our definition.

A TN is a 9-tuple, $M = (Q, \Sigma, P, \Gamma, \delta, \gamma, q_0, Z_0, f)$, where

(1) Q is a finite set of states corresponding to the states in the diagrams;
(2) Σ is a finite set of input symbols, and there is one symbol in Σ for each input token generated by the presentation component;
(3) P is a finite set of actions, which are the actions that label the states in the transition diagram;
(4) Γ is a finite set of push-down list symbols, and in our case there is one symbol in Γ for each state in Q and each symbol in Σ;
(5) δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to $Q \times \Gamma^*$, called the state transition function;
(6) γ is a mapping from Q to P, called the action function;
(7) $q_0 \in Q$ is the initial state of M;
(8) $Z_0 \in \Gamma$ is the initial symbol on the push-down list;
(9) $f \subset Q$ is the set of final states of M.

The configuration of a TN is an ordered pair (q, z) in $Q \times \Gamma^*$, where q is the current state of the TN, and z is the contents of its push-down list. Whenever the TN receives an input token from the presentation component, it moves to a new configuration and emits the name of an action. The new configuration and action are governed by the state transition function and the action function. If the TN has the configuration $(q, \alpha Z)$ and the input symbol a is received, then the new state and symbol on the top of the stack are given by $\delta(q, a, \alpha)$. The action emitted is given by $\gamma(q')$, where q' is the new state.

At the start of the dialogue the configuration of the TN is (q_0, Z_0). As input tokens are received, the TN moves from one configuration to another according to the state transition function. The TN recognizes the strings of input tokens that cause it to move from its initial configuration to one of the configurations (q, Z_0), where $q \in f$.

3.2 Context-Free Grammar Model

The context-free grammar model is based on context-free grammars, with an extension to cover the invocation of application actions. A context-free grammar model G is a 5-tuple $G = (N, T, R, P, S)$, where

(1) N is a finite set of symbols called nonterminals;
(2) T is a finite set of symbols called terminals, and there is one symbol in T for each of the input tokens produced by the presentation component;
(3) R is a finite set of symbols that correspond to the actions attached to the productions;
(4) P is a finite set of productions of the form

$$n \rightarrow \alpha, r,$$

where $n \in N$, $\alpha \in (N \cup T)^*$, and $r \in R$;
(5) $S \in N$ is the start symbol for the grammar.

A production can be viewed as a rule stating that the nonterminal on the left side can be replaced by the symbols on the right side. For example, if we have the production

$$X \rightarrow abc$$

and the string of symbols

$$\alpha X \beta,$$

then we can replace the nonterminal X by the string of symbols abc, giving the following string:

$$\alpha abc\beta.$$

This operation is called a derivation step and can be written in the following way:

$$\alpha X\beta \rightarrow \alpha abc\beta.$$

A sequence of derivation steps such as

$$\alpha_0 \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \alpha_n$$

can be abbreviated as

$$\alpha_0 \rightarrow^* \alpha_n.$$

The language, $L(G)$, described by a grammar G, is defined in the following way:

$$L(G) = \{x \mid S \rightarrow^* x, \text{ where } S \text{ is the start symbol of } G\}.$$

If G describes the dialogue control component of a user interface, then $L(G)$ contains all (and only) the legal sequences of user actions. A context-free grammar G can be used to produce a parser, which recognizes the sequences of user actions in $L(G)$.

Parsing algorithms can be divided into two groups: bottom up and top down. A top-down parsing algorithm begins with the start symbol of the grammar and expands it using the productions in the grammar. As the derivation steps are performed, the terminals on the right sides of the productions are matched against the tokens generated by the presentation component. The parse will continue as long as the terminals match the input tokens. More details on top-down parsing can be found in the standard textbooks on compilers, for example [3].

When a production is expanded in a top-down parse, the action attached to the production is emitted. A production is selected for expansion as soon as the first terminal that is able to be generated by its right side can be successfully matched against the current input token. Thus the action attached to a production will be executed when the start of a command is recognized.

A bottom-up parse works in the opposite direction. A bottom-up parse accumulates input tokens until they match the right side of a production. At this point the tokens are replaced by the nonterminal on the left side of the production, and the process continues. This replacement process continues until the start symbol for the grammar is reached. More details on bottom-up parsing can be found in [3].

As in the case of a top-down parse, the action attached to a production is emitted when that production is used in the parse. In the case of a bottom-up parse the production is not used until all the symbols on its right side have been recognized. Thus the action will be executed when the command represented by the production has been completely entered.

As can be seen from the above discussion, the point at which the action is used depends upon the parsing technique used. This is one of the main problems with using grammars as a notation for the dialogue control component. The description of the dialogue will depend upon the parsing algorithm used. In general, it is not easy to move a grammar-based description of the dialogue from one UIMS to another.

In practice, the full range of context-free languages is not used because of the inefficiency of their parsers. The largest set of languages that can efficiently be parsed are the deterministic context-free languages [2]. These are the languages that can be parsed by an LR(k) parser.

3.3 Event Model

The two major components of the event model are events and event handlers. In practice, event handlers are embedded in general-purpose programming languages. This gives the event model a considerable amount of power, but tends to hide its general properties. In order to characterize this dialogue model, an abstract version of it is constructed. This abstract version is independent of any particular programming language and has a relatively simple structure. This abstract model is called an abstract event system.

In an abstract event system all events have the same structure. An event E is a 3-tuple $E = (i, m, d)$, where

(1) $i \in I$, where I is a finite set of symbols that are used as names for the event handlers in the event system;
(2) $m \in M$, where M is a finite set of symbols that are used as event names;
(3) $d \in D$, where D is a domain (possibly infinite) that is used for event values. In this paper D is usually the integers.

In real event systems the event handlers are procedures written in a general-purpose programming language. In our abstract event system a real programming language would needlessly complicate the model. Instead, a simple language consisting of a finite set of registers and six simple statements is used. An event handler has a finite number of registers, each capable of storing one value from D or I. One register is assigned to each type of event handled by the event handler. The user interface designer can use the other registers to hold temporary results computed in the processing of an event, or to store a value used in the processing of several events.

Each type of event is processed by a separate procedure, called an event-handler procedure. The body of this procedure is a sequence of statements. Six types of statements, which represent the primitives of the event model, can appear in an event-handler procedure. The six statement types are as follows:

(1) ri := expression

This statement calculates a new value for one of the registers in the event handler. The expression consists of registers, constants, and operators. The expression cannot reference data values in other event handlers, or in the application.

(2) $I \leftarrow M$ ri

This statement creates a new event that has the name M and data value ri. This event is sent to the event handler I, which can be specified by either a constant or a register value.

(3) ri = create(template, $e_1, e_2, \ldots, e_n$)

This statement creates a new event handler on the basis of the event-handler template passed as a parameter. The behavior of this new event handler is defined by the template. The other parameters are the initial values for the registers of the new event handler. If there are not enough values to initialize all the registers, the unspecified registers are set to zero.

(4) destroy(ri)

This statement destroys the event handler identified by the value stored in ri.

(5) IF *condition* THEN *statement* ELSE *statement*

The IF statement allows for the selective execution of two sets of statements, depending upon the value of a condition. The condition can consist of registers, constants, operators, and logical connectives. IF statements can be nested.

(6) call proc($\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n$)

This statement is a call to one of the application procedures. The arguments to the procedure are expressions consisting of registers, constants, and operators. The application procedure cannot return a value to the event handler.

The above six types of statements are sufficient for our purposes. It is worth noting what cannot be done with these statements. There is no way of producing a loop within an event-handler procedure. There is also no way of producing a procedure or macro that can be called from inside an event-handler procedure. As a result, each event-handler procedure will terminate within a finite length of time. There is no way of constructing an infinite loop within an event-handler procedure. This simplifies the implementation of the event model, since a preemptive scheduler is not required.

An event handler EH is a 5-tuple EH = (m, r, Q, R, P), where

(1) m is the number of event types that are processed by this event handler;
(2) r is the number of registers in the event handler, with the restriction that $m \leq r$;
(3) $Q \in E^*$ is the event queue for the event handler. The events in Q have been sent to EH, but have not been processed yet;
(4) $R \in (D \cup I)^r$ is the set of register values for EH;
(5) P is a set of m event-handler procedures, with one procedure in this set for each type of event that can be processed by the event handler.

The configuration of an event handler is the contents of its event queue and the values of its registers. This configuration is written as (q, ρ), where $q \in E^*$ and $\rho \in (D \cup I)^r$. An event system ES is a finite set of event handlers. The number of event handlers in an event system can change dynamically as new event handlers are created and old ones are destroyed.

The behavior of an event system is governed by the following rules. The first rule is that, when an input token is received from the presentation component, it is converted into events that are added to the ends of the queues of the event handlers that have declared an interest in that token. The information in the event-handler templates is used to construct a table that maps input tokens into the corresponding events and the event handlers that are to receive these events.

The second rule deals with how the individual event handlers move from one configuration to another. The processing of an event is viewed as an atomic operation. That is, the configuration of the event handler does not reflect the execution of the individual statements in the event-handler procedures. If the current configuration of the event handler is (eq, ρ), the next configuration of the event handler will be (q', ρ'). In this new configuration ρ' is the register values after the execution of the event-handler procedure for the event e, and q' is of the form $q' = qr$, where $r \in E^*$. In this expression r represents the events that are sent by the event handler to itself. The execution of the event-handler procedure could also add events to the end of the event queues belonging to other event handlers in the system.

The third rule deals with the concurrent execution of the event handlers. Several event handlers can be executing in parallel. As stated above, the processing of an event is an atomic operation; an event handler can only process one event at a time. Also the events sent to an individual event handler are processed in the order in which they are sent. There are no constraints on the time ordering of events sent to different event handlers. This allows the dialogue control component to be distributed over several processors.

The standard way of implementing the event model is to use a preprocessor that converts programs in the event language into programs in a standard programming language. The event-handler procedures are written in the programming language that is the target of the preprocessor. As a result, the preprocessor only needs to translate the language components that are unique to the event language.

For some programming languages a run-time support library is required to handle the parts of the event language that are not easily supported in the base programming language. Examples of these features are scheduling the execution of event-handler procedures, sending events, and multiple instances of the same event handler. This approach to the implementation of event languages is described further in [8] and [19].

4. DESCRIPTIVE POWER

In this section two approaches are taken to the descriptive power of the three notations. The first approach is theoretical in nature, drawing on formal language theory. The second approach is more practical, showing how two of the models can be translated into the third.

A TN is essentially a deterministic push-down automaton (DPDA). The only difference between a DPDA and a TN is the action function γ. Since γ has no effect on the languages accepted by a TN, we can conclude that a TN has the same descriptive power as a DPDA. As mentioned in Section 3.2, only a subset of the context-free grammars are used in user interface managements systems. This subset is called deterministic context-free grammars.

It can be shown that the set of languages generated by deterministic context-free grammars and the set of languages recognized by DPDAs are the same (see [2] or [28] for a discussion of this point). This implies that the descriptive power of recursive transition network and context-free grammar models are the same. This observation is not quite correct. Consider a transition diagram where there are two paths leading from the start state. The first n input tokens on these paths are the same, but they differ on the $n + 1$ input token. For this type of diagram we can construct a DPDA that looks ahead $n + 1$ input tokens to determine the correct path to follow. This is essentially what an LR(k) parser does, by looking k symbols ahead in its input stream. Unfortunately, in the case of user interfaces, this approach cannot be taken, since there could be actions attached to the states (or arcs) before the $(n + 1)$st input token. These actions cannot be used when the user enters the corresponding input token (since we do not know which path we are on); therefore, the user is deprived of syntactic feedback. In practice, a restriction is placed on transition diagrams, so that at each state it is possible to determine the correct path given the current input token. Under this restriction recursive transition networks are equivalent to LL(1) grammars, which are a subset of the deterministic context-free grammars.

In practice an event model is embedded in a programming language. This gives the event model the descriptive power of a Turing machine. Thus the event model in this context has greater descriptive power than the other two models. It can be argued that this is an unfair comparison; therefore the event model defined in Section 3.3 will be used as the basis for our comparisons. In the next two sections algorithms for converting transition networks and grammars into event handlers are outlined. This shows that the event model has at least the same power as the other two models. In order to show that the event model is more powerful, a user interface that can be described in the event model, but not in the other two models, must be produced. The cut-and-paste example discussed in Section 2.3 can be used for this purpose.

In this example the cut-and-paste facility is part of a multifile editor. Each file has its own window and a set of event handlers that implement the basic editing functions within that window. Since each window has its own set of event handlers, the user can freely move from one window to another at any point in the dialogue. The cut-and-paste facility allows the user to move (or copy) text from one place to another within a window or between two windows. This facility can be invoked at any point, regardless of the states of the dialogues in the windows involved. In order to move a section of text, the user selects the text to be transferred, the cut or copy command from a menu, the new position for the text, and the paste command. The position at which the text is inserted can be selected at any point before the paste command is selected.

An event handler for the cut-and-paste feature is shown in Figure 11. This event handler processes five types of tokens. The cut, copy, and paste tokens are generated when the user selects the corresponding command from a menu. The select token is generated whenever the user selects a section of text within one of the editing windows. This token identifies both the text selection and the window where the selection occurred. The position token is generated each time the user selects a position. This token includes the window and the position (within that window) that the user selected.

Since the cut-and-paste facility can be invoked at any point in the editing session and its execution can overlap other editing operations, this type of dialogue cannot be described by transition networks or grammars. Other types of dialogues that are hard or impossible to describe with transition networks and grammars are described in [22].

The above theoretical result has two practical implications. First, if a UIMS only supports one design notation, the event model would be the best choice if range of coverage is one of the goals of the UIMS. Second, a UIMS based on the event model is capable of supporting all three models, since for each user interface described by recursive transition diagrams or a context-free grammar there is an equivalent description in the event notation.

The second implication can be made stronger by presenting efficient algorithms for converting recursive transition networks and context-free grammars into the event notation. These algorithms form part of the proof that the descriptive power of the event model is greater than the other two models. In the remainder of this section these algorithms are developed.

4.1 Converting Recursive Transition Networks

The conversion of a set of recursive transition diagrams is based on constructing an event-handler template for each diagram. An event handler is created from this template each time the diagram must be traversed and is destroyed when the traversal of the diagram is complete. Two parameters are passed to the event handler when it is created, the instance that called it and the state at the end of the arc in the calling diagram. When an event handler reaches a final state, it sends a "continue" event to the calling instance. The value of this event is the state in which the calling diagram is to continue processing.

The first step in the conversion algorithm is the calculation of the LEADING relation for each of the subdiagrams in the transition network. For a

```
EVENT HANDLER cut_and_paste;

 TOKEN
  cut Cut;
  copy Copy;
  paste Paste;
  select Select;
  position Position;

 VAR
  selection : text;
  owner : integer;
  current_selection : text;
  selection_owner : integer;
  current_position : point;
  position_owner : integer;

 EVENT Select DO {
  selection = Select.selection;
  owner = Select.owner;
 };

 EVENT Position DO {
  current_position = Position.position;
  position_owner = Position.owner;
 };

 EVENT Cut DO {
  owner <- "delete_selection";
  current_selection = selection;
  selection_owner = owner;
 };

 EVENT Copy DO {
  current_selection = selection;
  selection_owner = owner;
 };

 EVENT Paste DO {
  position_owner <- "add_selection" current_position current_selection;
 };

END EVENT HANDLER cut_and_paste;
```

Fig. 11. Event handler for cut and paste.

subdiagram d, let $L(d)$ stand for the set of strings in Σ that are recognized by d. That is, every string in $L(d)$ labels a path from the initial state to one of the final states of d. The relation LEADING is defined in the following way:

$$\text{LEADING}(d) = \{a \mid a \in \Sigma \text{ and } aS \in L(d)\}.$$

The relation LEADING for a given subdiagram is the set of input symbols that subdiagram is expecting when it is invoked. The procedure for calculating LEADING is similar to the one used to calculate the FIRST relation for context-free grammars (see [3]).

In the remaining steps the diagrams are considered one at a time, and an event-handler template is constructed for each diagram. The diagram that is being considered by the algorithm is called the current diagram.

In the second step of the algorithm a skeleton of the event-handler template for the current diagram is constructed. This skeleton contains all of the template, except for the event-handler procedures. In order to construct the skeleton, the token set for the current diagram must be constructed. This set is constructed in the following way. The token set is initialized to the empty set. Each of the arcs in the diagram is examined, and if the arc is labeled by an input token, that token is added to the token set. If the arc is labeled by a diagram name, all the tokens in the LEADING set for that diagram are added to the token set. Each token in the token set is used to produce a line in the TOKEN section of the event-handler template. These are the tokens that the event handler is interested in.

The transition diagrams for the polyline example (see Figure 5) are used to illustrate the conversion algorithm. For these diagrams the token sets are

```
token_set(main) = {button, move, finish}
token_set(command) = {move, backspace, button, finish}
```

The structure of an event-handler skeleton is shown in Figure 12. The event-handler template has the same name as the corresponding diagram. The two parameters to the event handler are the name of the event handler that invoked it and the number of the state at the end of the current arc in the invoking diagram. The local variable active indicates whether the event handler is currently processing input. If the value of this variable is 1, then input tokens are processed; otherwise, they are ignored. The local variable state records the current state of the diagram. This variable is initialized to the first state in the corresponding transition diagram. The third local variable is a temporary variable used to store the value returned by the create operator.

The third step of the algorithm is the construction of the event-handler procedures for the events corresponding to the tokens in the token set. Each of the event-handler procedures has the following basic structure:

```
EVENT $event_i$ DO {
  IF active = 1 THEN
    IF state = $n_1$ THEN
      .
      .
      .
    ELSE IF state = $n_2$ THEN
      .
      .
      .
    ELSE IF state = $n_m$ THEN
      .
      .
      .
    ENDIF
  ENDIF
}
```

There is one IF statement in the body of the procedure for each arc that is labeled by the corresponding token. In this case the procedure updates the state variable to the state at the head of the arc and executes any actions that are attached to that state. There is also an IF statement for each arc that is labeled by a subdiagram name that has the corresponding token in its leading set. In this case the procedure invokes the subdiagram and passes the event on to it. Applying this process to the main diagram results in the event-handler procedures shown in Figure 13.

The fourth and final step in the conversion algorithm is to construct the event-handler procedure for the continue event. The continue event signals the end of the traversal of a subdiagram that was invoked by the current diagram. The value of this event is the state at the head of the arc that invoked the subdiagram. The continue event handler has the following structure:

```
EVENT continue DO {
  active = 1;
  state := continue;
  IF state = $state_1$ THEN
    call procedure for $state_1$;
  ELSE IF state = $state_2$ THEN
    call procedure for $state_2$;
            .
            .
            .
  ELSE IF state = $state_n$ THEN
    call procedure for $state_n$;
  ENDIF
}
```

There is one IF statement in this procedure for each state that appears at the end of an arc labeled by a subdiagram. This IF statement calls the procedure that labels the head state.

The continue event-handler procedure for the main diagram is shown in Figure 14. After each of the diagrams has been considered, the conversion is complete.

4.2 Conversion of Context-Free Grammars

One approach to converting context-free grammars into event handlers is to have the event handlers simulate one of the standard parsing algorithms. The conversion algorithm presented in this section is based on the standard LL(1) parsing algorithm [3], which is a top-down parsing algorithm. The LL(1) parsing algorithm was chosen for the following two reasons. First, this algorithm has been used in at least one well-known UIMS [32]. Second, the LL(1) parsing algorithm is fairly simple and intuitive, but at the same time covers a wide range of grammars. The LR(1) parsing algorithms are more powerful but considerably more complicated. A more complicated parsing algorithm serves no purpose in this presentation.

The grammar shown in Figure 15 describes the same user interface as the set of transition diagrams shown in Figure 5. This grammar is used to illustrate the conversion of context-free grammars into event handlers.

```
EVENT HANDLER diagram_name(caller : I; head : integer);

 TOKEN
  token_1 event_1
  token_2 event_2
   .
   .
   .
  token_n event_n

 VAR
  int active = 1;
  int state = first_state;
  int temp;

END EVENT HANDLER diagram_name;
```

Fig. 12. Skeleton for event-handler template.

```
EVENT Button DO {
 IF active = 1 THEN
  IF state = 1 THEN
   state = 2;
   record first point;
  ENDIF;
 ENDIF;
};

EVENT Move DO {
 IF active = 1 THEN
  IF state = 2 THEN
   active = 0;
   temp = activate(command,this_instance,2);
   temp <- "Move" Move;
  ENDIF;
 ENDIF;
};

EVENT Finish DO {
 IF active = 1 THEN
  IF state = 2 THEN
   active = 0;
   temp = activate(command,this_instance,2);
   temp <- "Finish" Finish;
  ENDIF;
 ENDIF;
};
```

Fig. 13. Event-handler procedures for main diagram.

```
EVENT continue DO {
 active = 1;
 state = continue;
};
```

Fig. 14. Continue event handler for the main diagram.

The conversion algorithm is based on constructing an event-handler template for each of the nonterminals in the grammar. An instance of this event handler is created each time the corresponding nonterminal must be expanded in the parse. In general, the production for a nonterminal will have

main	→ button loop
loop	→ command loop I command
command	→ finish I move command I button continue
continue	→ backspace I command next
next	→ move I backspace

Fig. 15. Grammar for example user interface.

the following form:

$$\begin{aligned} A \rightarrow\ & \alpha_1 \\ |\ & \alpha_2 \\ & \vdots \\ |\ & \alpha_n \\ |\ & \epsilon \end{aligned} \tag{1}$$

The symbols $\alpha_1, \alpha_2, \ldots, \alpha_n$ represent nonempty strings of terminals and nonterminals, and the symbol ϵ represents the empty string. This production states that the nonterminal A can be replaced by the strings $\alpha_1, \alpha_2, \ldots,$ or α_n, or by the empty string. All the nonterminals will have a production of this form, except that the ϵ alternative may not be present.

The conversion algorithm consists of the following steps. First, the relations FIRST and FOLLOW must be computed. These relations have the following definitions:

$$\text{FIRST}(X) = \{a \mid a \in T, X \rightarrow^* aR, R \in (N \cup T)^*\}$$
$$\text{FOLLOW}(X) = \{a \mid a \in T, S \rightarrow^* \alpha X a \beta, \alpha, \beta \in (N \cup T)^*\}$$

The FIRST(X) relation is the set of terminals that can appear at the start of a string derived from the nonterminal X. Note the similarity between the FIRST and the LEADING relation for recursive transition networks. The FOLLOW(X) relation is the set of terminals that could follow X in some derivation in the grammar.

The FIRST and FOLLOW relations for the example user interface are shown in Figure 16. The algorithm presented in [3] was used to compute this relation.

The remaining steps of the conversion algorithm consider the nonterminals one at a time and construct the event-handler template for it. The nonterminal currently being considered by the algorithm is called the current nonterminal.

The second step consists of constructing a skeleton for the event-handler template. In order to construct this skeleton the token set for the event handler must be computed. This set contains all the tokens that the event handler is interested in. Assuming that the production has the form shown in Eq. (1), the token set can be computed in the following way. Each symbol in each alternative

```
FIRST(main) = { button }
FIRST(loop) = { finish, move, button }
FIRST(command) = { finish, move, button }
FIRST(continue) = { backspace, finish, move, button }
FIRST(next) = { move, backspace }
FOLLOW(command) = { finish, move, button, backspace }
FOLLOW(continue) = { finish, move, button, backspace }
FOLLOW(next) = { finish, move, button, backspace }
```

Fig. 16. FIRST relation for example user interface.

is examined. If the symbol is a terminal, then it is added to the token set of the production. If the symbol is a nonterminal, then all the symbols in its first set are added to the token set. If the production has an ϵ alternative, then all the symbols in the FOLLOW set for the nonterminal on the left side of the production are added to the token set.

The format for the skeleton of the event-handler template is essentially the same as the one used for recursive transition networks (see Figure 12). The name of the event-handler template is the same as the nonterminal it represents. The two parameters to the event handler are the name of the event handler that invoked it, and the current state of the invoking event handler.

The third step of the algorithm is to construct the event-handler procedures. First, the symbols on the right sides of the production are numbered. The first symbol in the first alternative is given the number 1, the second symbol is given the number 2, etc. This process continues until all the symbols have been numbered (the ϵ alternative is not numbered). These numbers, called the states of the production, are used to keep track of the current position within the parse. Next, the event-handler procedures are constructed, one at a time. There is one procedure for each of the tokens in the token set. The event-handler procedures have essentially the same format as the ones used for recursive transition networks.

The actions taken by the event-handler procedure depends upon the current state of the production. If the symbol at the current state is a terminal, the state advances to the next symbol in the production or returns to the calling production if the end of the production has been reached. If the symbol is a nonterminal, then the event handler corresponding to the nonterminal is invoked. Applying this process to the command production results in the event-handler procedures shown in Figure 17.

The fourth step in the conversion algorithm is constructing the continue event-handler procedure. The continue event is received when an event handler invoked by the current event handler has recognized its nonterminal. The processing of this event depends upon whether the corresponding nonterminal is the last symbol in the alternative. The value of the continue event is the state number of the nonterminal that invoked the event handler. The basic form of the continue event-handler procedure is

```
EVENT continue DO {
  state = continue + 1;
  active = 1;
  IF continue = $n_a$ THEN
    call procedure for alternative $a$;
    caller ← "continue" current_state;
    destroy(this_instance);
  ENDIF
    .
    .
    .
  IF continue = $n_z$ THEN
    call procedure for alternative $z$;
    caller ← "continue" current_state;
    destroy(this_instance);
  ENDIF
}
```

In the above procedure n_a through n_z are the state numbers of the nonterminals that appear at the end of the alternatives.

After each of the nonterminals have been processed by these steps, the conversion is complete.

4.3 Analysis of the Conversion Algorithms

Earlier in this section it was stated that there are efficient algorithms for converting recursive transition networks and context-free grammars into event handlers. There are two potential meanings for the word efficient in this context. The first meaning is that the algorithms can efficiently convert from one form of description to another. This aspect deals with the actual running time of the algorithm. The second meaning is that the event handler produced by the conversion algorithms is an efficient implementation of the dialogue. Both of these meanings are important in a UIMS and are discussed further in this section.

The efficiency of the conversion algorithms themselves can be approached in the following way. The computations of LEADING, FIRST, and FOLLOW are $O(N^2)$ in the worst case, where N is the number of transition diagrams or the number of alternatives in the grammar. The worst case is rarely realized in practice. The remaining steps of the algorithms are repeated M times, where M is the number of transition diagrams or the number of productions in the grammar. Each of these iterations involves a small number of traversals over the data structure representing the transition diagram or production. Thus this part of the algorithms is $O(M)$. The efficiency of these algorithms is quadratic in the worst case, and on average they will be close to linear.

The efficiency of the implementation must be measured against the techniques normally used to implement recursive transition networks and context-free grammars. A common approach to implementing both of these techniques is to use a form of table look up. Given the current state of the dialogue and the current input token, a table entry determines the next state of the dialogue and

```
EVENT Move DO {
 IF active = 1 THEN
  IF state = 0 THEN
   active = 0;
   temp = activate(command,this_instance,2);
   temp <- "Move" Move;
  ENDIF;
  IF state = 3 THEN
   active = 0;
   temp = activate(next,this_instance,3);
   temp <- "Move" Move;
  ENDIF;
 ENDIF;
};

EVENT Backspace DO {
 IF active = 1 THEN
  IF state = 0 THEN
   caller <- "continue" current_state;
   destroy(this_instance);
  ENDIF;
  IF state = 3 THEN
   active = 0;
   temp = activate(next,this_instance,3);
   temp <- "Backspace" Backspace;
  ENDIF;
 ENDIF;
};

EVENT Finish DO {
 IF active = 1 THEN
  IF state = 0 THEN
   active = 0;
   temp = activate(command,this_instance,2);
   temp <- "Finish" Finish;
  ENDIF;
 ENDIF;
};

EVENT Button DO {
 IF active = 1 THEN
  IF state = 0 THEN
   active = 0;
   temp = activate(command,this_instance,2);
   temp <- "Button" Button;
  ENDIF;
 ENDIF;
};
```

Fig. 17. Event-handler procedures for command production.

any actions that must be performed. This approach is quite fast, but tends to be space inefficient since the tables are fairly sparse.

The conversion algorithms essentially compile the table into the event handlers. In some dialogues this could result in a space saving. On receiving a token, the active event handler performs the operations that would be indicated by the table look up. The only operation that could be viewed as expensive is the creation of a new event handler. In our implementation the most expensive part of the creation operation is allocating memory for the event handler's local variables. This does not adversely affect the efficiency of the implementation. In practice, the user cannot tell the difference between a dialogue implemented by event handlers, and one implemented by special-purpose transition network or grammar software. Although the event-handler implementation cannot be as fast as special-purpose implementation techniques, it is not significantly slower.

In view of the above comments, the use of the event model as the basis for the dialogue control component of a UIMS will not significantly affect the execution speed of the resulting user interface or the system used to generate it.

5. CONCLUSIONS

In this paper we have investigated the three main dialogue models, which are transition networks, grammars, and events. Formal definitions of the dialogue models have been presented, along with algorithms for converting dialogue descriptions written in them into an executable form. It has been shown that the event model is the dialogue model with the greatest descriptive power. Efficient algorithms for converting recursive transition diagrams and context-free grammars into event handlers have also been presented.

What can we conclude from the above results? One conclusion that could be made is that a UIMS need not support recursive transition networks and context-free grammars, since an equivalent set of event handlers could always be constructed. This conclusion is oversimplistic, since it ignores the usability of the three notations. For example, some user interfaces are much easier to describe with recursive transition networks than with event handlers. The same thing can be said about context-free grammars. This suggests that a UIMS should support all three dialogue models.

A more useful conclusion from the above results is that the event model should form the basis for the internal representation used in the dialogue control component. The implementor of a UIMS only needs to implement run-time support for the event model; the other two models can be translated into this form. This gives the user interface designer the ability to choose the type of notation that best suits the application at hand (in some cases a combination of notations may be preferred). Whatever notation is chosen will be translated into the event-based internal form used by the UIMS. This approach has the dual advantage of allowing the user interface designer the choice of design notations. At the same time the UIMS implementor only needs to optimize the implementation of one design notation. An example of this type of UIMS is the University of Alberta UIMS [19].

REFERENCES

1. Aho, A. V., and Johnson, S. C. LR parsing. *ACM Comput. Surv. 6*, 2 (June 1974), 99–124.
2. Aho, A. V., and Ullman, J. D. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
3. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
4. Anson, E. The semantics of graphical input. In Siggraph '79 Proceedings, *ACM Comput. Graph. 13*, 2 (Aug. 1979), 113–120.

5. ANSON, E. The device model of interaction. In Siggraph '82 Proceedings, *ACM Comput. Graph. 16*, 3 (July 1982), 107–114.
6. BORUFKA, H. G., KUHLMANN, H. W., AND TEN HAGEN, P. J. W. Dialogue Cells: A method for defining interactions. *IEEE Comput. Graph. Appl. 2*, 5 (1982), 25–33.
7. CARDELLI, L., AND PIKE, R. Squeak: A language for communicating with mice. In Siggraph '85 Proceedings, *ACM Comput. Graph. 19*, 3 (July 1985), 199–204.
8. CHIA, M. S. An event based dialogue specification for automatic generation of user interfaces. M.Sc. thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1985.
9. COOMBS, M. J., AND ALTY, J. L., Eds. *Computing Skills and the User Interface.* Academic Press, London, 1981.
10. DENERT, E. Specification and design of dialogue systems with state diagrams. In *Proceedings of the International Computing Symposium* (Liège, Belgium). North-Holland, Amsterdam, 1977, pp. 417–424.
11. EDMONDS, E. A. Adaptive man–computer interfaces. In *Computing Skills and the User Interface*, M. J. Coombs and J. L. Alty, Eds. Academic Press, London, 1981.
12. EDMONDS, E. A., AND GUEST, S. P. An interactive tutorial system for teaching programming. In *Proceedings of the IERE Conference 36–Computer Systems and Technology* (Brighton, England, 1977). Institute of Electrical and Radio Engineers, London, pp. 263–270.
13. ELSHOFF, E. L., BECKERMEYER, R., DILL, J., MARCOTTY, M., AND MURRAY, J. Handling asynchronous interrupts in a PL/1-like language. *Softw. Pract. Exper. 4* (1974), 117–124.
14. FLECCHIA, M. A., AND BERGERON, R. D. Specifying complex dialogues in ALGEA. In *Proceedings of CHI and Graphics Interface '87* (Toronto, Canada, Apr. 5–9).
15. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
16. GREEN, M. A graphical input programming system. M.Sc. thesis, Dept. of Computer Science, University of Toronto, Toronto, Canada, 1979.
17. GREEN, M. Report on dialogue specification tools. *Comput. Graph. Forum 3* (1984), 305–313.
18. GREEN, M. Design notations and user interface management systems. In *User Interface Management Systems*, G. E. Pfaff, Ed. Springer-Verlag, Berlin, 1985, pp. 89–107.
19. GREEN, M. The University of Alberta user interface management system. In Siggraph '85 Proceedings, *ACM Comput. Graph. 19*, 3 (July 1985), 205–213.
20. GUEST, S. P. The use of software tools for dialogue design. *Int. J. Man-Mach. Stud. 16* (1982), 263–285.
21. HANAU, P. R., AND LENOROVITZ, D. R. Prototyping and simulation tools for user/computer dialogue design. In Siggraph '80 Proceedings, *ACM Comput. Graph. 14*, 3 (July 1980), 271–278.
22. HILL, R. D. Supporting concurrency, communications and synchronization in human–computer interaction—The Sassafras User Interface Management Systems. Special Issue on User Interface Software. *ACM Trans. Graph 5*, 3 (July 1986), 179–210.
23. JACOB, R. J. K. Executable specifications for a human-computer interface. In *Proceedings of the CHI'83 Human Factors in Computing Systems* (Boston, Mass., Dec. 12–15). ACM, New York, 1983, pp. 28–34.
24. KAMRAN, A. Issues pertaining to the design of a user interface management system. In *User Interface Management Systems*, G. E. Pfaff, Ed. Springer-Verlag, Berlin, 1985, pp. 43–48.
25. KAMRAN, A., AND FELDMAN, M. B. Graphics programming independent of interaction techniques and styles. *ACM Comput. Graph. 17*, 1 (1983), 58–66.
26. KAY, A., AND GOLDBERG, A. Personal dynamic media. *Computer 10* (Mar. 1977), 31–41.
27. KIERAS, D., AND POLSON, P. G. A generalized transition network representation for interactive systems. In *Proceedings of the CHI'83 Human Factors in Computing Systems* (Boston, Mass., Dec. 12–15). ACM, New York, 1983, pp. 103–106.
28. LOMET, D. B. A formalization of transition network systems. *J. ACM 20*, 2 (Apr. 1973), 235–257.
29. NEWMAN, W. M. A system for interactive graphical programming. In *Proceedings of the Spring Joint Computer Conference* (Atlantic City, N.J., Apr. 30–May 2). Thompson, Washington, D.C., 1968, pp. 47–54.
30. NEWMAN, W. M. A high-level programming system for a remote time-shared graphics terminal. In *Pertinent Concepts in Computer Graphics*, M. Faiman and J. Nievergelt, Eds. University of Illinois Press, Urbana, Ill., 1969.

31. OLSEN, D. R. Pushdown automata for user interface management. *ACM Trans. Graph. 3*, 3 (1984), 177-203.
32. OLSEN, D. R., AND DEMPSEY, E. P. SYNGRAPH: A graphic user interface generator. In Siggraph '83 Proceedings, *ACM Comput. Graph. 17*, 3 (July 1983), 43-50.
33. PARNAS, D. L. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th National ACM Conference* (San Francisco, Calif., Aug. 26-28). ACM, New York, 1969, pp. 379-385.
34. PFAFF, G. E., Ed. *User Interface Management Systems.* Springer-Verlag, Berlin, 1985.
35. SCHULERT, A. J., ROGERS, G. T., AND HAMILTON, J. A. ADM—A dialogue manager. In *Proceedings of the CHI'85 Human Factors in Computer Systems* (San Francisco, Calif., Apr. 14-18). ACM, New York, 1985, pp. 177-183.
36. SIBERT, J., BELLIARDI, R., AND KAMRAN, A. Some thoughts on the interface between user interface management systems and application software. In *User Interface Management Systems,* G. E. Pfaff, Ed. Springer-Verlag, Berlin, 1985, pp. 183-192.
37. TANNER, P., MACKAY, S. A., STEWART, D. A., AND WEIN, M. A multitasking switchboard approach to user interface management. In Siggraph '86 Proceedings, *ACM Comput. Graph. 20*, 4 (1986), 241-248.
38. TEN HAGEN, P. J. W., AND DERKSEN, J. Parallel input and feedback in dialogue cells. In *User Interface Management Systems,* G. E. Pfaff, Ed. Springer-Verlag, Berlin, 1985, pp. 109-124.
39. VAN DEN BOS, J. Definition and use of higher-level graphics input tools. In Siggraph '78 Proceedings, *ACM Comput. Graph. 12*, 3 (Aug. 1978), 38-42.
40. VAN DEN BOS, J., PLASMEIJER, M. J., AND HARTEL, P. H. Input-output tools: A language facility for interactive and real-time systems. *IEEE Trans. Softw. Eng. SE-9*, 3 (1983), 247-259.
41. WOODS, W. A. Transition network grammars for natural language analysis. *Commun. ACM 13*, 10 (Oct. 1970), 591-606.

Received July 1986; revised December 1986; accepted December 1986