McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

User Interface Description Formalisms

Vipul Jain

ACAPS Technical Memo 67
April 23, 1994

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

A user interface description is an abstract model that is used to describe the structure of a dialogue between a user and an interactive computer system. It forms the basis of notation that are used in user interface management systems. In this paper three classes of description formalisms are investigated. These classes are single-threaded representations (state-transition-diagram and grammar based specifications), multi-threaded representation (events, state charts based specifications) and concurrent dialogue representations (Petri nets and Process Calculus based specifications). Formal definitions of all the models are presented along-with algorithms for converting the notions into executable forms, which are used in various User Interface Management Systems. An evaluation method is presented and the implication of the results for the design and implementation of UIMs are also discussed.

Contents

List of Figures

1 Introduction

It is useful to be able to write a description of the user interface of a computer system before building it, because the interface designer can thereby describe and study a variety of possible user interfaces without having to code them. Such a description should describe precisely the user-visible behavior of the interface, but should not constrain its implementation. Specification techniques for describing the user-visible behavior of conventional user interfaces without reference to implementation details are gaining currency.

If the description formalism is sort of language and itself can be executed or compiled, it can also serve as the basis for a user-interface management system (UIMS). To be useful, a UIMS needs a convenient and understandable way for the user-interface designer to describe the desired interface. The choice of specification language is thus at the heart of design of UIMS. UIMSs have been built using BNF or other grammar-based specifications [OD83], state-transition-digram based specifications [FR82, Jac83, SRH85, WS85] , programming-language-based specifications [Kas82], frames [Hay85], flow-diagrams [YH85], object-oriented approach [Gre85, Lib85, SHB86] and other models [Ans82, BLSS83]. With the recent advancement in technology like Multi-media applications etc, concurrent dialogues have become necessary when user interfaces support more than one stream of output for simultaneous presentation to the user. The representation techniques under research for representing concurrent dialogues are use of variations of Petri nets and Process algebra.

The formalism 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 formalism must be capable of describing widest possible range of user interfaces. The range of a description formalism can be measured in two ways. The first measure of the range is descriptive power of the formalism. The descriptive power of the formalism is the set of user interfaces that can be described by the notation. Determining the descriptive power of a description formalism 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 description formalism is by its usability power. The usability power of a description formalism is the set of user interfaces that can easily be described by the formalism. The focus of this survey paper [Jai94] is to present various description formalism their properties, and how these formalisms are used in describing user interfaces.

2 Description Formalisms

The three classes of description formalisms are, single-threaded, multi-threaded and concurrent. Informal descriptions of different models are given in this section and formal definitions are presented in the next section. In this section we will see the informal description of each and their relevance with user interface behavior specification.

2.1 Single-threaded description formalism

State transition networks and context-free grammars are two most popular models under single-thread description formalisms.

2.1.1 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 labelled by an action(an input token) that the user can perform. The arc 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 labelled 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 state. A sequence of user actions is accepted if they label the arcs on a path through diagram.

A simple example of a dialogue described by a transition diagram is shown in Figure 1.

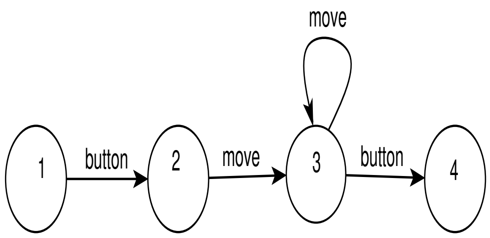


Figure 1: A simple transition diagram.

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

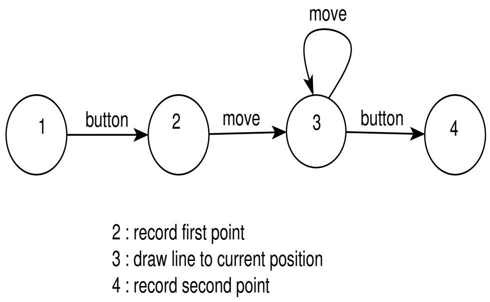


Figure 2: Example transition diagram with program actions.

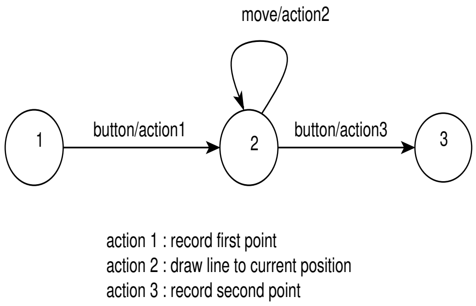


Figure 3: Example transition diagram with actions on arcs.

Program actions can also be attached to the arcs. In this case a program action is executed when an 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,a 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.

There are two problems with 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 sub-diagrams. A sub-diagram is a complete transition diagram that can be invoked from another diagram. These sub-diagrams can be viewed as procedures or subprograms. Any arc in a diagram can now be replaced by a call to one of the sub-diagrams. An arc labeled by a sub-diagram name will be traversed, when the user enters a sequence of actions that causes the sub-diagram to be traversed, from its start state to one of its final states. An extended definition of arc traversal that covers sub-diagram can now be produced.

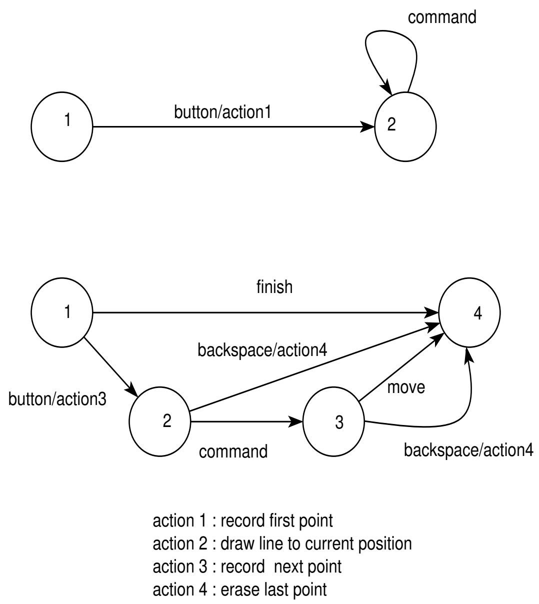
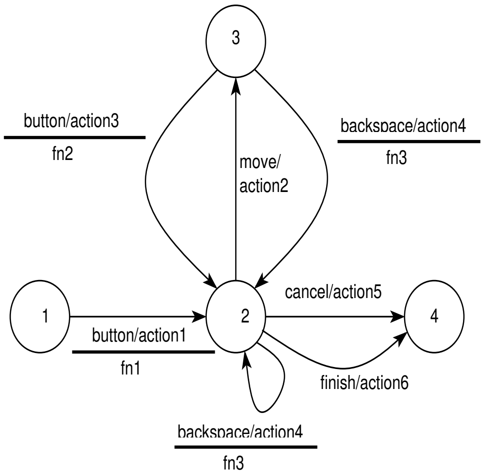


Figure 4: Transition diagram for polyline example.

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 sub-diagram name and user performs action that cause the sub-diagram to be traversed.

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

Augmented transition networks (ATN) are an extension of recursive transition network [Woo70]. 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 function 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.




```
action 1 : record first point
action 2 : draw line to current position
action 3 : record  next point
action 4 : erase last point
action 5 : erase polyline
action 6 : 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 ) ;
```

Figure 5: Polyline dialogue with cancel.

It is possible to construct context-sensitive dialogue with augmented transition networks (this is not the case for other transition networks). For example in a database query application, the register value 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 a 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 [Kam85, KP83] or [SBK85]). Jacob [Jac83] has used a notation similar to augmented transition networks for specifying and prototyping user interfaces. The interactive pushdown automata suggested by Olsen [Ols84] also resemble augmented transition networks.

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

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 next state. Another solution is to use a wild card label. This arc label will match any user action that does not match any other user actions. A wild card arc leads to a state where error recovery is attempted. Another approach for handling unexpected user actions is Olsen's pervasive states and transitions [Ols84]. In this approach a transitions diagram can have another transition diagram associated with it , 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. The 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 [New68, New69]- and Paras [Par69]. Since then, transition networks have been the basis of a large number of UIMSs (e.g., Kamran and Fledman [KF83] and Schulert et al [SRH85])

The first use of recursive transition networks in user interfaces appears to be the work of Denert [Den77]. The SYNICS system is another early example of use of recursive transition networks. Experience with the use of both transition networks and recursive transition networks is presented in Kamran [Kam85] and Sibert et al. [SBK85] .

2.1.2 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 programmer to describe the dialogue between the user and the computer. This 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 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 a grammar define the language employed by the user in his or her interaction with the computer. An example of this type of description is shown in Figure 7. This example is same as the one used for transition networks.

```
line   --->  button end_point

end_point  --->  move end_point
                 | button
```

Figure 6: Context-free grammar for rubberband 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 }
```

Figure 7: Rubber band line example with program actions.

The grammar in Figure 6 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 diagram, program actions are attached to the states in the diagram to indicate the program's responses. A similar approach can be used in context-free grammars; program actions can be attached to the each of productions in the grammar. These program actions are performed when the production is used in the parse of a user's input. One of the major parse depends upon the parsing algorithm used. In the case of bottom-up parse, a production is used when all the symbols on its right side have been recognized. In the case of top-down parse, the production is used when the first terminal that could be generated by the right side is encountered. By assuming that a top-down parse is used, the example in Figure 6 can be augmented by the program actions to obtain the grammar shown in Figure 7. The grammar has been modified in order to produce the program actions at the appropriate place in the dialogue. This modification introduces holders for the program actions. Most of the grammar-based UIMs provide some notational mechanism for invoking actions in the middle of productions (Olsen and Dempsey [OD83] describe one such possible mechanism). In these notations the introductions 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. A number of extensions to context-free grammars have been proposed and implemented in some UIMSs [OD83, VDB78]. Most of these extensions deal with error recovery, undo processing, and control over the order of the parse. It should be noted that existing UIMs do not use the full power of the context-free grammars. The grammars used in these systems are restricted to the subset of the 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 in user interfaces is hard to determine. In the mid-1970s a number of research groups used commonly available parser generators, such as YACC [AJ74] to produce the user interfaces to graphics programs. In most of the 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 (r.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 does 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 systems [EG77, Gue82]. Another early example of the use of context-free grammars is the work of Hanau and Lenorovitz [HL86].

The SYNGRAPH system developed by Olsen has a number of interesting properties [OD83]. 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 content of menus and when devices must be acquired and released.

Input-output tools [TMSW86, VDB78] and *dialogue cells* [BKTH85, Chi85] take a different approach to the specification of the 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). Actions are performed when the end of the production is reached, the echo produced for the interaction, and the value returned by the production.

2.2 Multi-threaded Description formalisms

In a *multi-threaded 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. Event handling and State-charts are the two formalism categorized under multi-threaded description formalism class .

2.2.1 Event

The *event* formalism is based on the concept of input events that are 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 when one of the cursor buttons is pressed or released.

```
EVENT HANDLER line;

        TOKEN
                button  Button;
                move Move;

        VAR
                int state;
                point first,last;

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

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

        INIT
                state = 0;

        END EVENT HANDLER line ;
```

Figure 8: Event handler for the rubber band line example.

The events are placed on a queue when they are generated, and the application program removes these 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 the input devices.

The event formalism for User Interfaces 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 are generated inside of dialogue control component. The programmer is free

```
EVENT HANDLER  polyline_cancel;

          TOKEN
                    button  Button;
                    move Move;
                    backspace Backspace;
                    cancel Cancel;
                    finish Finish;
          VAR
                    int state;
                    point_list : list of point ;
                    point _count : integer ;

          EVENT Button DO {
                    IF state == 0 THEN
                              point_list = currnet 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 first to current position;
                    ENDIF;
          };
          EVENT Finish DO {
                    return (point_list);
                    deactivate(self);
          };
          EVENT Backspace DO{
                    IF point_count > 1 THEN
                              remove last point from the poit_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 ;
```

Figure 9: Event handler for polyline with cancel.

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 Small-talk

[GR83]). 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 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 to 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 is created it remains 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 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 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 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 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 event is from the process itself. An event handler can send an event to one of the other event handlers in the dialogue control components. 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 communication between event handlers.

An event handler for the rubber band line example is shown in Figure 8. A notation similar to that is presented in Green [Gre85] is used in this example. The definitions of the line event handler is divided into four sections. The first section declares the token 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.

An event handler can be constructed for the polyline example with the cancel extension. This is shown in Figure 9.

The event formalism was motivated by early work on Small-talk [KG77]. One of the first implementations of this model was the work of Green [Gre79]. Two other UIMs that support the event formalism are Sassafras [Hil86] and ALGEA [FB87]. Anson [Ans79] 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 [Ans79]. Cardelli and Pike [CP85] have produced a language called Squeak for processing the input from mice and keyboards. This language is based on processes and messages between same purpose as events. Tanner et al. [TMSW86] have also described a multiple process and message-based system to support graphics and user interfaces. This system is based on 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.

2.2.2 State-charts

States and Events are a priori a rather natural medium for describing the dynamic behavior of a complex system. A basic fragment of such a system is a *state transition*, which takes the general form "when event α occurs in a state A, if condition C is true at the time, the system transfers to state B". Finite state machines and their corresponding state transition diagrams are the formal mechanism for collecting such fragments into a whole. State transition diagrams as described before are simply directed graphs, and it is generally agreed that a complex system cannot be beneficially described in this naive fashion, because of the unmanageable , exponentially growing multitude of states, all of which have to be arranged into a "flat" unstratified fashion, resulting in an unstructured, unrealistic, and chaotic state diagram [EA83, Par69]. To be useful, a state/event approach must be modular, hierarchical and well-structured. *State-charts* constitute a visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality (i.e concurrency) and refinement for moving easily back and forth between levels of abstraction [Har87].

The kernel of the approach is the extension of conventional state diagram by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components. Two essential ideas enabling this extension are provision for "deep" descriptions and the notion of orthogonality. In a nutshell, one can describe it as..

state-charts = state-diagrams + depth +
orthogonality + broadcast-communication

Figure 10 shows a state-chart. Here round rectangles are used to denote states at any level, using encapsulation to express the hierarchy relation. Arrows are allowed to originate and terminate at any level. There are three states A,B, and C. Event γ occurring in state A

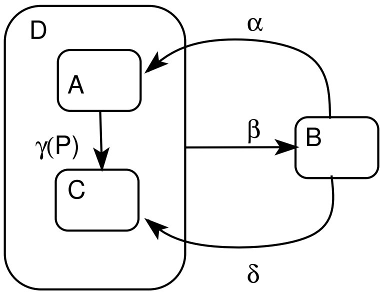


Figure 10: State-Chart

transfers the system to state C, but only if condition P holds at the instant of occurance. Now, since event β takes the system to B from either A or C, we can cluster the latter into a new super-state D and replace the two β arrows by one.

The semantics of D is then the *exclusive-or* (XOR) of A and C; i.e. to be in state D one must be either in A or in C, and not in both. Thus, D is really an *abstraction* of A and C. The state D and its outgoing arrows β arrows thus capture a common property of A and C, namely that β leads from them to B. The decision to let transitions that leave a super state, such as the β in figure 10, stands for transitions leaving all the sub-states turnout to be highly important, and is the main way state-charts economize in the number of arrows.

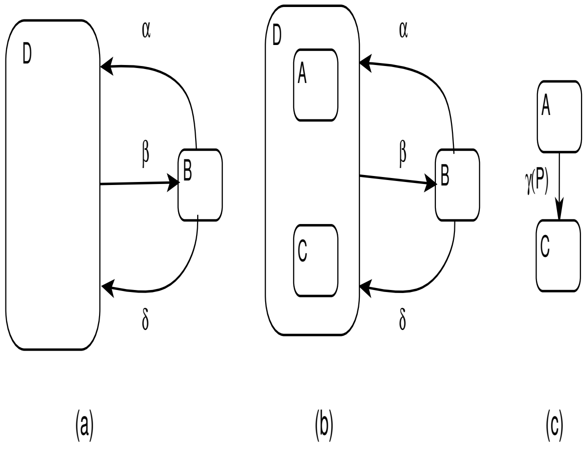


Figure 11: Clustering, Refinement, and Abstraction in State-Chart

Figure 10 might be approached from a different angle: first we might have decided upon the simple situation of Figure 11(a), and then state D could have be refined to consists of A and C, yielding Figure 11(b). Having made this refinement, however the incoming α and β arrows become under-specified, as they do not say which of A or C is to be entered. Extending them to point directly to A and C, respectively, does the job, and if the γ transition within D is added, one indeed obtains Figure 10. Thus clustering, or abstraction, is a bottom-up concept and refinement is top-down one; both give rise to the or-relationship between a state's sub-states.

The AND decomposition on states captures the property that, being in a state, the system must be in all of its AND components. The notion used in state-charts is the physical splitting of a box into components using dashed lines.

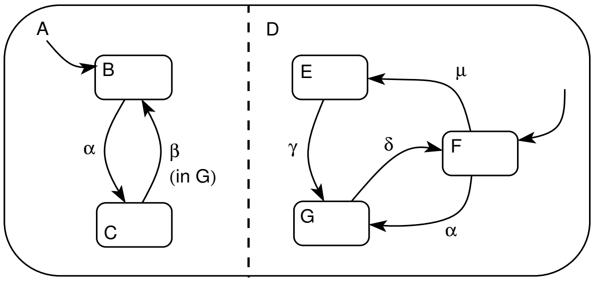


Figure 12: Orthogonality in State-chart

Figure 12 shows a state Y consisting of AND components A and D, with the property that being in Y entails in some combination of B or C with E,F or G. We say that Y is *orthogonal product* of A and D. The components A and D are no different conceptually from any other super-states; thus they have defaults, internal transitions, etc. Entering Y from the outside, in the absence of any additional information, is actually entering the combination (B,F) by the default arrows. If event α then occurs, it transfers B to C and F to G simultaneously, resulting in the new combined state (C,G). This illustrates a certain kind of synchronization: a single event causing two simultaneous happening. If on the other hand, μ occurs at (B,F) it affects only the D component, resulting in (B,E). This in turn, illustrates a certain kind of *independence*, since the transition is the same whether the system is in B or in C in its A component. Both behaviors are part of the orthogonality of A and D, which is the term used here to describe the AND decomposition.

Recursion, though seen little in practice, can be introduced into state-charts, just as augmented-transition-networks. This can be done by specifying the name of a separate state chart, the present one being a special case, on a transition arrow. Such an extension would require a notation for terminal states, say a double state boundary.

Probabilism can also be added to state-charts. The way to do this is to allow nondeterminism, and then to specify a bias on the coin to be tossed when it arises. This, in fact would really be a tantamount to considering "Markov-charts" which would be classical Markov chains what state-charts are to the classical transition diagrams of finite-state automata.

The concept of State-charts was first introduced by Parnas [Par69] in 1969. Since then state-charts were being used in various applications like synthesis of digital systems [EA83] and specification and verification of communication protocols etc. In description of UIs it was David Harel who gave the complete formalism [Har84] and used them in System analysis and prototyping tool Statemate. The Statemate uses state-charts to describe the behavior of system under development (SUD). It provides tow complimentary views of SUD. These are *structural view*, which describes the physical modules and *functional view*, which describes the data flow and activities.

2.3 Concurrent dialogue based formalisms

When the user interfaces support more than one stream of output for simultaneous presentation to a user, as in the case of Multi-media applications, concurrent dialogue become necessary. A concurrent dialogue based formalism requires that some form of multitasking or parallel processing be used to produce the concurrent dialog threads. It is required that the concurrent dialog specification technique to be able to

- start a new thread that executes concurrently with existing threads,
- terminate an existing dialog thread that is executing concurrently with other existing dialog threads
- synchronize two or more existing threads so that one doesn't get ahead of the other.

The representation formalism used for these are Process Calculus and variations of Petri-nets.

2.3.1 Process Calculus

The formalism is based on the theory of Concurrent Communicating Systems (CCS) [Mil89], which incorporates an algebraic language for processes, a deduction system for reasoning about processes based on a set of equations, and a class of equivalence relations.

The key point in this formalism is the notion of an *agent*. An agent models and describes a specific part of a complex system. Communication is achieved by *actions* that agent may perform. Each agent is capable of performing an action to interact with its neighboring agents, which occurs independently and concurrently with other agent's actions. The capability of an agent to perform a specific action is modeled as a communication port. There are two kinds of ports: input ports and output ports. Communication between agents is achieved by linking input and output ports of agents. The basic constructors for agent expressions are:

- The inactive agent 0
- Constant A *(the Agent A)*
- Prefix a.E *(do Action a and behave like Agent E)*
- Summation E1 + E2 *(behave like Agent E1 or Agent E2)*
- Composition $E1 \mid E2$ *(Agent E1 composed with Agent E2. The Agents may proceed independently or may interact through complementary ports)*
- Restriction E L *(Actions in the set L, are visible only within the scope of Agent E)*
- Relabelling E[f] *(Port renaming of Agent E according to function f. It is used for communicating agents which share no complementary ports)*
- Substitution E/X *(Substitute X by E)*
- Recursion fix(X = E) *(the Agent X such that X = E)*

Equation are used to represent process behavior. For example the equation $A = b.B + c.C$ is interpreted as: *Agent A is capable of performing action b and then behaving like agent B, or A is capable of performing action c and then behaving like agent C.*

The composition operator links two agents into a system in which the agents may proceed independently or may interact through complementary ports. Handshakes through complementary ports are represented by a special action denoted by the perfect action τ.

The restriction operator (|) is used to hide ports from other agents that may seek communication.

The relabeling operator [*f*] allows for renaming of communication ports. This operator is used for communicating agents with no complementary ports.

An example of modelling system behavior is shown in Figure 13 [Mil89]. The system is a simple machine for selling chocolates. The behavior of the machine is defined in terms of 5 ports - (2p, 1p, big, little, collect)- as follows

$$V = 2p.big.collect.V + 1p.little.collect.V + 1p.1p.big.collect.V$$

The meaning of above equation is that in order to buy a big chocolate, you have to insert a *2p* coin, press the button marked *big*, and *collect* the chocolate from the tray along with all possible alternatives.

Thus the behavior of a system is defined in terms of its entire communication capabilities, or alternatively, in terms of what is observable in such a system.

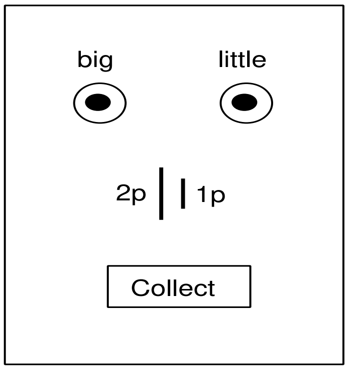


Figure 13: Example modelling with process algebra

2.3.2 Petri Nets

The formalism based on Petri net consists of a set of places, transitions, and directed arcs.

Places are denoted by ovals and represent the presentation of information to the user or entry of information from the user. Associated with each place is a unit of time. If the activity is associated with input from the user, the time is determined by how long the user takes to enter the information. If the activity associated with the place is the presentation of information to the user, the time denotes the length of the presentation. This time may be fixed (as in the time for replying a video or audio clip) or dependent upon other activities (such as the time it takes the user to select and enter an option from a menu presented by another place). *Transitions* are denoted as vertical bars and represent synchronization points among parallel dialogs. Directed arcs connect places and transitions. It is convenient to think of control flow tokens which flow through the directed arcs connecting transitions and places. Transitions act as type of "gate" which controls the starting of activities associated with places. After it has received a control flow token from each of its input places, the transaction places a control flow token on each of its output places. Places cause semantic token to be exchanged with the user and/or underlying application programs. After receiving a control flow token, a place does the following:

1. The place exchanges semantic tokens with the user and/or underlying application programs.
2. After the place has completed all of its activities, it places a control flow token on each transition.

Figure 14 illustrates part of an interactive Petri net which consists of four places and three transitions. Place P_1 denote the display of a video clip to the user, place P_2 denotes the presentation of an audio clip to the user, place P_3 denotes the display of a menu of the user, and P_4 denotes the user's selection of one of the options presented on the menu. The transitions

t_1 denotes the simultaneous beginning of P_1 and P_2. The transition t_2 denotes the ending of both P_1 and P_2 and the simultaneous beginning of P_3 and P_4. The transition t_3 denotes the ending of both P_3 and P_4.

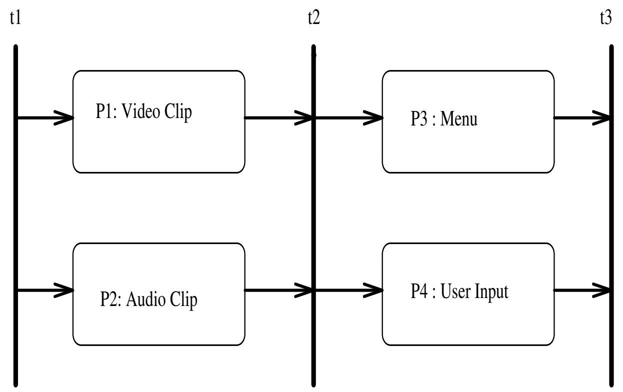


Figure 14: A Petri net.

Initially, transition t_1 places control flow tokens in places P_1 and P_2. Both P_1 and P_2 fire at the same time, with P_1 displaying a video clip and P_2 replaying an audio clip. When P_2 completes its audio clip replay, it also places its control flow token in t_2. When transition t_2 has received control flow tokens from P_1 and P_2 it places control flow tokens on P_3 and P_4. Place P_3 displays a menu of options to the user and places a control flow token on t_3. After the user selects a menu choice, it places it control flow token on t_3. When both the control flow tokens from places P_3 and P_4 have been placed on t_3 the concurrent dialog stops.

Thus we see that basic formalism provides a fair mechanism for modeling and study of behavioral properties of concurrent systems but there are some problems with this formalism. These are:

- Unavailability of strong hierarchical decomposition.
- Provides only one level of concurrency.
- Kind of high-level encompassing events that are applicable at once in many lower level places are not naturally specifiable.

A solution to above is to introduce depth into the definition of *place* in a Petri net, with transitions connected to places at all levels. There are two extensions suggested for these.

1. Colored Petri nets (CP-net)
2. Predicate/Transition Nets

The *Colored Petri net* [**?**] combine the power of formal inscriptions (i.e mathematical expressions) with the clarity of graphical models to provide concise representations of complex system. CP-net formal inscriptions include the following:

- **Color Set** definitions specify the types of tokens in the model. Tokens can be complex data objects (e.g., tuples or lists). The color set definitions are analogous to type declarations in programming languages. Each place in a CP-net has an associated color set which constrains the tokens allowed to occupy it.

- **Markings** specify the multi-set of tokens occupying each place, represented as an algebraic sum.

- **Arc Expressions** characterize the multi-set of tokens that will "move" along an arc when the associated transition occurs. Arc expressions may contain variables, predefined or user-defined functions and boolean selectors, along with usual operators. For any occurance of transition, the variables in the associated arc expressions are bound to particular values.

- **Guards** restrict the allowable bindings of the variables occurring in the arc expressions.

The hierarchical representation of CP-net [PKR88]contains a set of sub-nets (each of which may contain sub-nets, and so forth). This provides a set of features:

- hiding of details in consistent way.

- separation into well defined components.

- reusable components.

- support of both top down and bottom up development strategies.

- strong graphical expressive capabilities.

These features constitutes a formalism which provides both modularity and manageability.

First use Petri-nets in UI was made by Shew Yoo in 1989 [SY89] in his project KBSE (Knowledge Base Software Environment). This supported concurrent programming based on Object-Oriented Knowledgebase. In 1991 David Scott and Richard Furuta used Petri-nets in modelling the behavior of a Hypertext System [FS89, SF91, SFR92]. They used petri-nets for specification of presentation objects when a hypertext is browsed. Petri-net also provided modelling of synchronization of simultaneous traversals of separate paths through the Hypertext. Jensen, Shapiro and Albrecht developed hierarchical Colored Petri- nets based design and simulation system [**?**, PKR88].

In this section formal definitions of the description formalisms are presented. The main reason for producing these definitions is to provide both a formal framework for comparing all the formalism and 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 application 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.

2.4 Single-threaded description formalisms

2.4.1 Transition Networks

The development of formal definition of the transition network formalism 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 network 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 the transitions between these states. The input alphabet is the set of input tokens that can be there 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) in 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 x Σ 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 STN is slightly different from that of finite-state machine. The configuration of 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 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 sub-diagrams in TNs. In a recursive transition network, when an arc labeled by a sub-diagram name is encountered, control is transferred to the sub-diagram. When the traversal of the sub-diagram is complete, control transfers back to the node at the end of the arc invoking the sub-diagram. Since sub-diagram 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 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 actions 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 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 $q \in f$.

2.4.2 Context-Free Grammar

The context-free grammar formalism 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 such as

$$\alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \alpha_n$$

can be abbreviated as

$$\alpha_0 \stackrel{*}{\rightarrow} \alpha_n,$$

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

$$L(G) = \{X \mid s \xrightarrow{*} x,\ where\ S\ is\ the\ start\ symbol\ of\ G\}$$

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 algorithm 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 terminal match the input tokens. More details on top-down parsing can be found in the standard textbooks on compilers, for example [ASU76].

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 token until they match the right side of the production. At this point the tokens are replaced by the nonterminals 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 [ASU76].

As in the case of a top-down parse, the action attached to a production is emitted when the production is used in the parse. In the case of 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 successfully 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 application of the dialogue from one UIMS to another.

2.5 Multi-threaded description formalisms

2.5.1 Event Formalism

The two major components of the event formalism are events and event handlers. In practice, event handlers are embedded in general purpose programming languages. This gives the event formalism a considerable amount of power, but tends to hide its general properties. In order to characterize this description formalism in Green [Gre85].

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

1. m is a number of event types that are processed by this event handler;
2. r is 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 w 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 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 $(q', \rho\prime)$. In this new configuration $\rho\prime$ is the register value 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; and event handler can only process one event at a time. Also the event sent to an individual event handler are processed in the order in which they are sent to different event handlers.

The standard way of implementing the event formalism 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 a 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. Example of these features are 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 [Chi85, CA81].

2.5.2 State-charts

The state-chart formalism turns out to be quite a challenge when it comes to proving formal semantics, much more so than simple finite-state automata. The main difficulty is not in the depth of states or the orthogonality constructs themselves; these (as claimed in [Har84]) can be dealt with by a translation into ordinary automata, although even there delicate problems arise. Consider Figure 15. The semantics must be able to conclude that when α occurs in (A,B), the system ends up in (C,D), but that one more α leads not to (E,F) but, nondeterministically, either to (E,H) or to (G,F). The more difficult problems arise with the introduction of events and conditions are generated within the state-chart itself, and are sensed in orthogonal components.

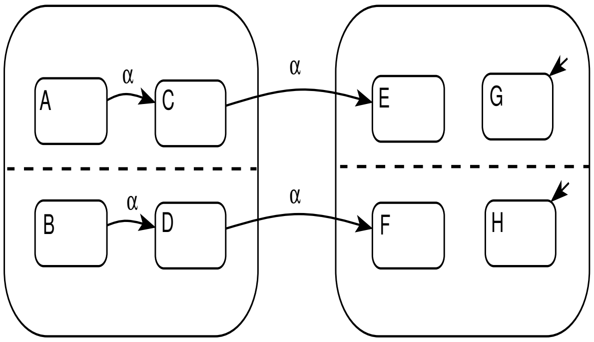


Figure 15: Transitions in state-charts

To a semantic this is obvious. The problems with concurrency stem from the cooperation mechanism: shared variables and their access, communication over channels, handshake rendezvous', etc. In this respect, state-chart employ a broadcast communication mechanism. One part generates an event (by an action that is placed, say, to the right of a "/" on a transition), and all other parts sense it, acting in response if so specified. Clearly, upon sensing such an event, another component might generate a new event, causing yet others to be generated.

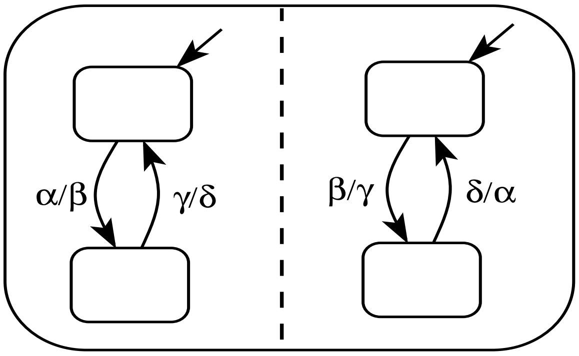


Figure 16: Cycles in State-chart

Cycles, like that of Figure 16, have to be dealt with by the semantics, presumably rendering them undefined.

Not only the events generated by actions cause problems, but also those generated by the very dynamics of the state-chart. We allow state-charts to refer to condition "in state", and to the events that signal changes in that condition, "entered state" and the "left state". At first sight these seems to be very well defined, the first as a time span and others as instants in time. However Figure 17 illustrates a situation in which it is not clear what the outcome should be. Starting in (A,C,E), if α occurs, should we end-up in (B,C,E), (B,D,E), (B,C,F), or (B,D,F)?. Intuition says, perhaps, (B,C,F), but a formal semantics must supply all the answers.

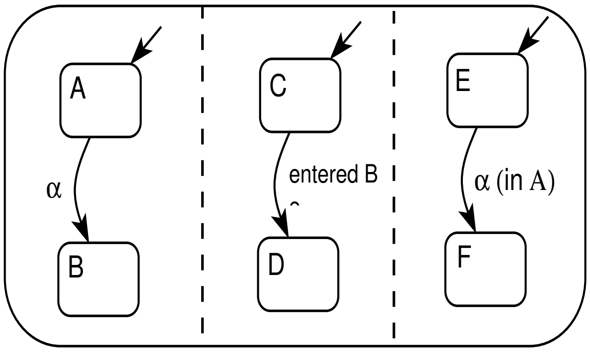


Figure 17: Non-determinism in State-charts

The state-chart is given by a state tree together with a set of transitions, each of which is an ordered pair of configurations (a *source* and a *target*) labelled with some legal $\alpha(P)/S$ combination. Defaults and history entrances are appropriately included. The heart of the semantics of [HPSS87] is the definition of a function *nextstep(X,C,E)*, which, for a full basic configuration X, a set of (external) conditions C, and a set of simultaneous (external) events E, provides the set of net possible full basic configurations.

The semantics assumes that each step represent some time interval in which new internal events may be generated and responded to. The main steps in defining *nextstep(X,C,E)* are the following. First, all *relevant* transitions are selected. This involves analyzing their *trigger* (the part of their label to the left of the "/") to see whether indeed it is triggered by E, and their source configuration to see whether it is ancestoral to a sub-configuration of X. For each such transition, the semantics takes off separately, analyzing the consequences of applying it (such as the consequences of the action part of its label), and finding new transitions, that become relevant as a result. This process continues until it closes off, reaching a stable and consistent situation. Then the configuration X is replaced by an appropriate $X\prime$, which is obtained by applying the overall effect just determined to the appropriate ancestoral configuration, and then "diving" downwards, using default entrances to reach a full basic configuration. The set of all such $X\prime$ is the final result of *nextstep*. The process of the stable and consistent situation is the tricky part, and details can be found in [HPSS87].

2.6 Concurrent dialogue based formalisms

2.6.1 Process Calculus

The notion of lablled transition is used to give the formal meaning to basic process algebraic language.

$$(S, T \xrightarrow{t} \ t \in T)$$

Which consists of a set S of states, a set T of *transition labels*, and a transition relation $\xrightarrow{t} \subseteq S \times S$ for each $t \in T$.

Let S be ξ, the agent expressions. T be *Act*, the actions. Then the semantics for ξ consists in the definition of each transition relation $\xrightarrow{\alpha}$ over ξ.

The transition of each composite agent is defined in terms of the transitions of its component agent or agents. For example the transition $A \xrightarrow{\alpha} A'$ infers $A \mid B \xrightarrow{\alpha} A' \mid B$;

The general rule which permits this is

$$\frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$$

.

There are one or more transition rules associated with each combinator, and one associated with Constants. Each rule has a *conclusion* and zero or more *hypotheses*. The conclusion is transition of an agent expression consisting of the combinator applied to one or more components, and the hypotheses is transition of some of the components. The complete set of transition rules is given below. The name **Act, Sum, Com, Res, Rel and Con** indicate that the rules are associated respectively with Prefix, Summation, Composition, Restriction, Relabelling and with Constants.

$$\textbf{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \textbf{Sum}_{\mathbf{j}}\ \frac{E_j \xrightarrow{\alpha} E_j'}{\Sigma_{i\in I}E_i \xrightarrow{\alpha} E_j'}(j\ \in I)$$

$$\textbf{Com}_{\mathbf{1}}\ \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \textbf{Com}_{\mathbf{2}}\ \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$\textbf{Com}_{\mathbf{3}}\ \frac{E \xrightarrow{l} E' F \xrightarrow{\overline{l}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

$$\textbf{Res}\ \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}(\alpha, \overline{\alpha} \notin L) \qquad \textbf{Rel}\ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\textbf{Con}\ \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}(A\ \stackrel{def}{=}\ P)$$

the rule for Summation can be read as follows: if any one summand E_j of the sum $\Sigma_{i\in I}$ has an action, then the whole sum also has that action. Finite summation, which is enough for many practical purposes, can be presented in a more convenient form. If I=1,2 then we obtain two rules for $E_1 + E_2$, by setting j = 1,2:

$$\frac{E_1\ \xrightarrow{\alpha}\ E_1'}{E_1 + E_2\ \xrightarrow{\alpha}\ E_1'} \qquad \frac{E_2\ \xrightarrow{\alpha}\ E_2'}{E_1 + E_2\ \xrightarrow{\alpha}\ E_2'}$$

Also, $0 \stackrel{def}{=}\ \Sigma_{iin0}E_i$; since I = 0 in this case, there is no rule for 0, and this reflects the fact that 0 has no transitions.

The set of rule is complete as there is no transition except those which can be inferred by the rules. The justification for transition of any agent expression in the form of an inference diagram can also be set out in which it annonates each inference with the name of the rule which justifies it.

There does not appear to be a "best" formalism to describe all user interface scripts. Some formalisms are better for user interface behavioral aspects, while other are better for usability aspects. In this section a criteria [Lar93]is discussed to evaluate each of the description formalism. And later in Figure 18 a comparison of all is presented. This evaluation does not cover the comparison of concurrent dialogue based formalism as all the formalism are still under research and haven't been explored much in context of User Interfaces.

2.7 Sequencing,multi-threading, and concurrency

State transition systems are useful for describing scripts in which user-entered information tokens are ordered. Multiple alternative ordering of information tokens can be represented by

1. Explicitly describing each information token order as a separate path or

2. Using OR grouping state charts to represent multi-threading or separate sequences.

Approach 1 leads to large and complex, state transition systems containing many nearly redundant states. Approach 2 leads to fewer states which are grouped in complicated ways.

Like state transition systems, *grammars* are useful for scripts in which user-entered information tokens are ordered. Multiple, alternative ordering of information tokens can be described in two ways:

1. multiple productions with alternative sequences of the same symbols on the right-hand side or

2. use of the "$A \mid B$" notation on the right-hand side of a production to indicate that either "A" or "B" may be entered first.

Multi-threaded techniques are frequently used to describe scripts in which users may enter information tokens in arbitrary orders. Ordering on the entry of information tokens can be imposed by encoding the ordering as part of multiple event handlers. Because most direct manipulation user interfaces do not require, and even avoid sequencing, multi-threaded techniques are frequently used for direct manipulation user interfaces.

2.7.1 Understandability

State transition systems are easy to understand and comprehend by most designers. Large state transition systems, however, may be intractable for many humans. Approaches for dealing with potentially large number of states and transactions include techniques like

1. replacing segments of state transition diagram by sub-diagrams

2. clustering closely related states into branches of state tree

3. merging states, and using super-nodes.

The notion of *grammar* is more concise and tighter (in the sense of taking up less space) than state transition systems. While this avoids large transition systems which are difficult for a human to comprehend quickly, the compactness of a grammar notion requires cognitive processing for human understanding. In grammars, the distributed nature of productions make it difficult for some user interface designers to visualize valid dialogues.

In the multi-threaded technique, a script's specification is distributed among several separate event handlers. It may be difficult for the dialogue designer to mentally fit these event handlers together into a complete script making it difficult for them to debug and modify scripts.

2.7.2 Sound formal basis

State transition systems have a solid theoretical basis. State transition systems have been extensively used in lexical analyzers and various types of control systems.

Grammars have a solid theoretical basis for compiler theory.

Multi-threaded techniques have some formal basis in the areas of structured programming (and concept hiding), abstract data types (where the implementation of the data types is encapsulated and hidden from users of the data type), and object-oriented programming. However, a general methodology for constructing multiple and concurrent agents is still a research issue.

2.7.3 Re-usability

State transition systems can be partitioned into subsystems, which can be reused in other dialogues.

In *grammars*, sets of productions used in scripts can be reused in other scripts. Script designer must take care when reusing productions to verify the names of the variables modified by actions are consistent with the new script and that there is no conflict in the names of symbols used in the productions.

Re-usability is one of the strong points for *multi-threaded* approach. Event handlers can be easily extracted from one script and reused in another. The dialogue designer should examine each message which can be sent by the object and verify that those messages can be responded to in the system to which the object is ported.

2.7.4 Executable

Both *state transition systems and grammars can be interpreted. This is especially desirable during script development and debugging. State transition systems and grammars can also be compiled and optimized for faster execution. This may be desirable for scripts which will be frequently used. Techniques from compilers are useful for compiling grammars.*

A few experimental multi-threaded execution engine exist [Hil86, FB87]. There are several object-oriented execution engines that may be used to implement multi-threading scripts.

2.7.5 Familiarity

Many dialogue designers sketch screen appearances on a sheet of paper and then they sequence the sketches. This "story board" approach is similar to specifying state transition system. Of the various representation techniques these are probably the easiest for dialog designers to learn and use.

Grammars may be difficult for many user interface designers, who will need additional training to design and manipulate scripts described using grammars.

While some dialogue designers are very enthusiastic about *multi-threaded approaches* others may find these designers fundamentally different from traditional programming techniques with which they are more familiar. In general, object-oriented approaches are becoming popular in the user interface community, especially for direct manipulation user interfaces.

2.7.6 Perspicuity

Because of the graphical nature of state transition systems, it may be possible for dialogue designers to identify subsystems common to several scripts, leading to the identification of common script fragments. It may also be possible to recognize dialogue styles using this formalism. But there is no such complete formalism available as yet.

For *grammars*, transformations on AND/OR trees can be used to place dialogues into a canonical form, making it easy to identify common script fragments.

2.7.7 Inconsistent and incomplete specification detection

For *state transition systems*, there exist algorithms to detect some types of incompleteness. It is possible to detect some states that can never be reached. It is also possible to detect some states that can never be exited.

For *grammars*, utilities can be built which analyze grammars to detect inconsistent and incomplete specifications.

The big drawback of *multi-threaded* is that inconsistent and incomplete specifications may exist. Without actually executing all possible scenarios, these incomplete specifications may not be discovered.

2.7.8 Measurability

It may be possible to measure the complexity of a script by counting the number of states in a *state transition system* which must be executed to allow for the user to accomplish a task.

For *grammars* [Rei84] has suggested that the number of production is propositional to the complexity of the dialogue. Because many different sets of productions can describe the same dialogue, it will be necessary to adopt a standard form for productions before using this approach to compare two alternative scripts. It is conceivable that the type of production may also be inductive of the complexity of the dialogue.

The number of event handlers might serve as a complexity measure for multi-threaded models. However, these approaches may only approximate the complexity.

2.7.9 Backup and recovery

With *state transition systems*, a state may have an explicit transition to previous state which reverses the actions of the current state. In this situation, users must explicitly enter the command which is the inverse of the most recently executed command. Some state transition execution engine maintain a pushdown stack of recently visited states with both the unmodified and modified values of the variables changed. Users may enter successive undo commands to reverse the actions of states.

Some execution engines based on *grammars* also maintain a pushdown stack of recently executed productions with the unmodified values of variables changed by those productions. Users may enter successive undo commands to reverse the actions of these productions.

In theory, *multi-threaded* systems can be designed to be reversible. They can also be designed to maintain a pushdown stack so that the users can enter successive undo commands. In practice, this is seldom done.

3 Conclusion

Traditional programming languages are often used to express scripts. They are being extended to support additional user interface features. In general, state transition systems or grammars are used to describe scripts in which the information tokens to be entered by the user are ordered. Multi-threaded techniques are used to describe scripts in which the user enters information tokens in arbitrary orders. In concurrent dialogue based formalisms Petri nets might be considered as one of the best-understood solutions. They are graphical and precise, and come complete with an impressive body of research. Milner's CCS formalism is also accumulating a large body of research, involving variations, formal semantics and detailed comparisons to other systems. Among the main characteristics of CCS are the adoption of recursion as main control structure, the availability of good level of concurrency, and renaming and aliasing flexibility provided by the use of variables for signals and communication channels. In fact, CCS is emerging as something of a yardstick, against which to measure the features, capability and power of new languages and formalisms to programming and specifying concurrency.

Therefore surrounded by such a variety of formalism it can not be said off hand that such and such formalism is the best for behavior description of a UI. It depends mainly on the

Criteria	**State Transition**	**Grammars**	**Events**
Sequncing, Multithreading & Concurrency	Good when information token are ordered. Multiple ordering	Good when information token are ordered. Multiple ordering	Used when iformation tokens are unordered. Event handlers
Understandability	Easy to understand	Compact notation. Requires	Difficult. - script making - debugging and modifications
Sound Formal basis	Good Theoratical basis. Used in lexical	Good Theoratical basis. Complilers.	Object Oriented paradigm provide the formalism. - Concept hiding
Re-usability	Good. Partioining of a dialogue script	Good. Script or portions of it can be	Another feature of Object Oriented Paradigm.
Familiarity	Sequnce of sketch drawing similar to specifying State Transition	Requires additional training	Different from traditional programming approach. OOP
Inconsitent and incomplete specification detection	Algorithms exists for incomplete and unreachable states detection	Utilities available for detecting inconsistencies.	Difficult to detect.
Measurablity	Measurable by Number of states.	Difficult to measure due to complexity of Productions.	Measurable by number of event handlers.
Backup and Recovery	Good and multiple ways to do it. - explicit path	Pushdown automata can be used for recovery.	Another feature of Object Oriented Paradigm.

Figure 18: A Comparision.

domain of application and the trade-off between descriptive power and usability of a formalism while choosing one for a UIMS or application framework.

References

[AJ74] A.V Aho and S.C Johnson. Lr parsing. *ACM Comput. Surv.*, 6(2):99–124, June 1974.

[Ans79] E. Ansen. The semantics of graphical input. *ACM Computer Graphics*, 13(2):113–120, Aug 1979. In Siggraph'79 Proceedings.

[Ans82] E. Ansen. The device model of interaction. *Computer Graphics*, 16(3):107–114, July 1982.

[ASU76] A.V Aho, R. Sethi, and J.D. Ullman. *Compiler Principles Techniques and Tools.* Addison-Wesley, Reading, Mass., 1976.

[BKTH85] H.G Borufka, H.W Kuhlmann, and P.J.W TEN HAGEN. Dialogue cells: A method for defining interactions. *IEEE Computer Graphics Appl*, 2(5):25–33, 1985.

[BLSS83] W. Buxton, M.R. Lamb, D. Sherman, and K.C Smith. Squeak:a language for communcating with mice. *Computer Graphics*, 17(3):35–42, July 1983.

[CA81] M.J Coombs and J.L. Alty. *Computing Skills and the User Interface.* Academic Press,London, 1981.

[Chi85] M.S. Chia. An event based dialogue specification for automatic generation of user interfaces, 1985. Dept. of Computer Sc., University of Alberta, Edmonton.

[CP85] L Cardelli and R Pike. Squeak:a language for communicating with mice. *ACM Computer Graphics*, 19(3):199–204, july 1985. In Siggraph'85 Proceedings.

[Den77] E.A. Denert. Specification and design of dialogue systems with state diagrams. In *Proceedings of the International Computing Symposium*, pages 417–424, 1977. Liege,Belgium.

[EA83] M.D. Edwards and A. Aspinall. The synthesis of digital systems using asm design techniques. *Computer Hardware Desciption Languages and their Applications*, pages 55–64, 1983. North-Holland, Amterdam.

[EG77] E.A. Edmonds and S.P. Guest. An interactive tutorial system for teaching programming. In *Proceedings of the IERE Conference 36-Computer Systems and Technology*, pages 263–270. Institute of Electrical and Radio Engineers,London, 1977. Brighton,England,1977.

[FB87] M.A. Flecchia and R.D. Bererson. Specifying complex dialogue in algea. In *Proceedings of the CHI and Graphics Interface'87*, pages 33–47, 1987. Toronto,Canada Apr. 5-9.

[FR82] M.B. Fledman and G.T. Rogers. Toward the design and development of style dependent interactive system. In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference*, pages 111–116. ACM,Newyork, 1982. Gaithersburg,Md., Mar. 15-17.

[FS89] Richard Furuta and P. David Stotts. Separating hypertext content from structure in trellis, 1989.

[GR83] A Goldberg and D Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, Reading, Mass., 1983.

[Gre79] M. Green. The graphical input programming system, 1979. Dept. of Computer Sc., University of Toronto, Toronto.

[Gre85] M. Green. The university of alberta user interface management system. *ACM Computer Graphics*, 19(3):205–213, July 1985. In Siggraph'85 Proceedings.

[Gue82] S.P Guest. The use of software tools for dialogue design. *Int. J. Man-Mach. Stud.*, 16:263–285, 1982.

[Har84] D Harel. A visual approach to complex systems. Technical report, Department of Applied Mathematics, The Weizmann Institute of Science, 1984.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hay85] P.J. Hayes. Executable interface definitions using form-based interface abstractions. *In Advances in Human-Computer Interacion*, pages 161–189, 1985.

[Hil86] R.D. Hill. Supporting concurrency, communications and synchronization in human-computer interaction-the sassafras user interface management system. *ACM Tran. Graph.*, 5(3):179–210, July 1986.

[HL86] P.R. Hanau and D.R Lenorovitz. Prototyping and simulation tools for user/computer dialogue design. *ACM Computer Graphics*, 14(3):271–278, July 1986.

[HPSS87] D Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of state-charts. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science*, 1987.

[Jac83] R.J.K. Jacob. Using formal specification technique for describing human-computer interface. *Communications of ACM*, 26(4):259–264, April 1983.

[Jai94] Vipul Jain. User interface specification languages. Technical report, School of Computer Science, McGill University, 1994.

[Kam85] A. Kamran. Issues pertaining to the design of a user interface management system. *In User Interface Management System*, pages 43–48, 1985.

[Kas82] D.J Kasik. A user interface management system. *Computer Graphics*, 16(3):99–106, July 1982.

[KF83] A. Kamran and M.B. Fledman. Graphics programming independent of interaction techniques and styles. *ACM Computer Graphics*, 17(1):58–66, 1983.

[KG77] A Kay and A Goldberg. Personal dynamic media. *Computer*, 10:31–41, Mar 1977.

[KP83] D. Kieras and P.G. Polson. A generalized transition network representation for interactive systems. In *Proceedings of the CHI'83 Human Factors in Computer Systems*, pages 103–106. ACM,Newyork, 1983. Boston,Mass., Dec. 12-15.

[Lar93] James.A. Larson. *Tools for Building Interactive User Interfaces.* Prentice-Hall Inc., 1993.

[Lib85] H. Liberman. There's more to menu systems than meets the screen. *Computer Graphics*, 19(3):181–189, July 1985.

[Mil89] R Milner. *A Calculus of Communicating Systems.* Prentice Hall Inc., 1989.

[New68] W.M. Newman. A system for interactive graphical programming. In *Proceedings of the Spring Joint Computer Conference*, pages 47–54. Thompson,Washington,D.C.,, 1968. Atlantic City, N.J., Apr. 30-May 2.

[New69] W.M. Newman. A high-level programming system for a remote time-shared graphics terminal. In *Pertinent Concepts in Computer Graphics*, pages 47–54. M.Faiman and J.Nievergelt,Eds. University of Illinois Press, Urbana, Ill., 1969.

[OD83] D.R. Olsen and E.P. Dempsey. Syngraph:a graphical user interface generator. *Computer Graphics*, 17(3):42–50, July 1983.

[Ols84] D.R. Olsen. Pushdown automata for user interface management. *ACM Tran. Graph.*, 3(3):177–203, 1984.

[Par69] D.L. Parnas. On the use of transition diagram in the design of a user interface for an interactive computer system. In *Proceedings of ACM Conf.*, pages 379–385, 1969.

[PKR88] Huber P., Jensen K., and Shapiro R.M. Hierarchies in coloured petri nets. Bonn, June 1988. Tenth International Conference on Application and Theory of Petri Nets.

[Rei84] Phyllis. Reisner. *Human Factors in Computer System*, chapter Formal grammar as a tool for analyzing ease of use: some fundamental Concepts, pages 53–78. Ablex,NewYork, 1984.

[SBK85] J. Sibert, R. Belliardi, and A. Kamran. Some thoughts on the interface between user interface management systems and application software. *In User Interface Management System*, pages 183–192, 1985.

[SF91] P. David Stotts and Richard Furuta. Dynamic adaptation of hypertext structure, 1991.

[SFR92] P. David Stotts, Richard Furuta, and J. Cyrano Ruiz. Hyperdocuments as automata: Trace-based browsing property verification, 1992.

[SHB86] J.L. Sibert, W.D. Hurley, and T.W. Bleser. An object-oriented user interface management system. *Computer Graphics*, 20(4):259–268, Aug 1986.

[SRH85] A.J. Schulert, G.T. Rogers, and J.A. Hamilton. Adm-a dialog manager. In *Proceedings of the ACM SIGCHI Human Factors in Computer System Conference (Sanfransisco, Calif.,)*, pages 177–183. ACM,NewYork, April 1985.

[SY89] P.C.Y Sheu and S. Yoo. A knowledge-based software environment (kbse) for designing concurrent processes. *International Journal of Human-Computer Interaction*, 1(2):161–185, 1989.

[TMSW86] P. Tanner, S.A. Mackay, D.A. Stewart, and M. Wein. A multitasking switchboard approach to user interface mangement. *ACM Computer Graphics*, 20(4):241–248, Aug 1986. In Siggraph'86 Proceedings.

[VDB78] J. Van Den Bos. Definition and use of higher level graphics input-output tools. *ACM Computer Graphics*, 12(3):38–42, Aug 1978.

[Woo70] W.A.. Woods. Transition network grammars for natural language analysis. *Communications of ACM*, 13(10):591–606, Oct 1970.

[WS85] A.I Wasserman and D.T. Shewmake. The role of prototypes in the user software engineering (use) methodology. *In Advances in Human-Computer Interacion*, pages 191–209, 1985.

[YH85] T. Yunten and H.R. Hartson. A supervisory methodology and notation (superman) for human-computer system development. *In Advances in Human-Computer Interacion*, pages 243–281, 1985.