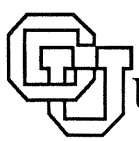
An Execution Model for Demonstration-Based Visual Languages

Wayne Citrin

CU-CS-610-93 September 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

An Execution Model for Demonstration-Based Visual Languages

Wayne Citrin

CU-CS-610-92

September 1992

Wayne Citrin
Dept. of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309-0425

citrin@soglio.colorado.edu

tel: 1-303-492-1688
fax: 1-303-492-2758

An Execution Model for Demonstration-Based Visual Languages

Wayne V. Citrin

Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309

citrin@cs.colorado.edu

ABSTRACT

An abstract execution model for demonstration-based and example-based visual languages is presented. The formalism, causality graphs, are a simplified version of Petri nets, and possess similar execution properties. We show how programming, compilation, and execution of demonstration-based languages may be expressed as the construction and analysis of causality graphs. The model provides a basis for a theory of demonstration-based languages, and leads to a number of suggestions for the design and implementation of these languages.

1. Introduction

Over the past decade, as graphic workstations have become more widely available, numerous visual languages and environments, which allow programming with pictures instead of, or in addition to, text, have been designed and implemented. Because of the added flexibility in representation of the manipulated entities and the operations upon them afforded by graphics, visual programming is especially suited for a *direct manipulation* model[17] in which the user carries out operations on the underlying data by "directly manipulating" the visual analogues of the data. The file remove operation on the Apply Macintosh user interface, in which an icon representing the file is moved to an icon representing a trash can, is a perfect example of this.

In order for users to be able to write programs using direct manipulation languages, rather than simply using them as command shells, language designers have turned to example-based programming[10]. In its simplest form, example-based programming is simply a scripting facility in which the system "observes" the manipulations and plays them back on command. Programming by Rehearsal[8] is an example of such a system.

Scripting systems are of only limited generality and usefulness, and attempts have been made to generalize such systems to create general programs from specific scenarios. In these systems, the user works through one or more scenarios, and the system generalizes a program that re-enacts the scenarios given the proper initial conditions, and hopefully acts out the intent of the programmer in other situations. Some systems, MetaMouse[12] and Cara system[6] for example, use heuristics to deduce the programmer's intentions. The former system creates programs to manipulate graphical

objects based on prototype scenarios involving mouse operations. The latter system attempts to deduce communications protocol specifications from a set of prototype diagrams. It is often the case that an example-based system based on heuristics gets it wrong, and, in the case of Cara, extensive programmer intervention is necessary to assure that the programmer's meaning is captured.

To avoid this problem, other systems incorporate programmer intent into the example scenarios; the programmer indicates both *what* happens and *why* it happens. Myers' example-based system for programming user interfaces, Peridot[13], requires the programmer to specify the relation between action and result; for example, the relation between the movement of a scroll bar slider and the scrolling of the associated window. MFD[5], an extension of Cara, avoids wrong guesses by requiring that the user specify the precise enabling conditions for protocol actions. The MFD diagrams contain all the information needed to form a correct protocol specification. This mode of programming has been called *programming-with-example*[14].

It would be extremely useful to possess a formal framework through which these and other languages could be described. Such a framework would allow formal, unambiguous definition of the systems. It would allow us to identify those language features that are common to most or all demonstrational languages, and those which are dependent on the particular demonstrational systems. It might suggest possible implementation strategies. Finally, it would provide a common conceptual framework for reasoning about demonstrational languages.

Some work has been done on underlying execution models of visual languages, but all of it has involved iconic languages[2, 9, 19]. In iconic visual languages, pictures (known as *icons*) representing data or operations are combined to create *iconic sentences*. The meaning of the component icons, combined with their spatial relationships, assigns a meaning to the sentence. Iconic sentences directly describe a definite sequence of operations, however; example-based systems that provide a generalization from a set of specific scenarios require a very different model. In this paper, we present a model for example-based languages, *causality graphs*, based on Petri nets. Precedence graph models related to causality graphs have been used in the specification of concurrent systems[15, 18], but they are also well suited to the specification of actions and their motivations as required by programming-with-example systems.

In developing our model, we first present a simple causality graph model. We then increase the power of the model by adding timing and negation considerations, and then apply the model to an example in the MFD programming language. Finally, we offer some suggestions for the design and implementation of demonstration-based languages based on the causality-graph model.

2. Basic causality graph model

2.1. Introduction

We model demonstration-based languages as sets of *events* and *causes*. The programmer demonstrates a scenario by providing the system with a sequence of events comprising the scenario. The programmer either explicitly specifies the cause of each event, or the system deduces the cause through heuristics, or there is some combination of the two. We see the archetypal demonstrational system as possessing a front end,

usually a graphical editor, that generates a stream of events and associated causes which it transmits to a module performing the tasks (demonstration, compilation, and execution) described in this paper. Execution generates another stream of events and causes, which the front end displays to the user.

An event is an abstraction of any change in system state. It may be an input or output event (a mouse operation, or the movement of a box on the screen, or the transmission of a message in a network, for example), or a change in internal state. Each event has associated with it a *type* and an *action*. If two distinct events have the same event type, they are both considered to be instances of that event type. Actions associated with events are programs of arbitrary complexity. Two events of the same type have the same associated action, although since the action may include choices based on system state, the actual effect of two "identical" actions may be different. Event types are used to generalize the information gathered from the demonstration scenario, while actions are used to assign executable semantics to events.

Each event has a cause, which is a (possibly empty) set of other events. Intuitively, a given event occurred because all the events in its cause occurred. Walking through a demonstration scenario is the process of constructing a network of such events and causes. Compilation is the generalization of this network into a rule base whose rules reflect each set of causal relationships generalized to cover event types, rather than specific events. Execution of the program involves constructing a new causal network based on the rules derived from the prototype network. These processes will be formalized in the following sections.

It should be noted that the causality model we present does not explicitly specify disjunctions. In a specific concrete scenario, an event has a specific cause, which is another event occurring in the scenario. The paradigm as it is generally realized does not allow for the expression of alternate causes for a given event. However, we are able to derive disjunctions through generalization to event types. If one event of type *a* is caused by events of types *b* and *c*, and another event of type *a* is caused by events of types *d* and *e*, we may say that an event of type *a* is caused by events of types *b* and *c*, or by events of types *d* and *e*.

It should also be noted that negation and timing considerations are not included in the basic model. They are necessary for any complete model, but require special considerations and will be discussed in a later section.

2.2. Definition of causality graphs

A *scenario* consists of a causality graph (defined below), a state (representing the state of the demonstration), and an integer representing the current value of the system's global clock. Any particular scenario instance represents the progress of a demonstration at a given point in time. In this paper, we will concentrate on the causality graph component; alterations to state and clock value will generally be generally be referred to implicitly.

A *causality graph* C is a pair (V,E), where V is the set of *events* and E is the set of directed *causality edges* (v_1,v_2), where v_1 and v_2 are both events in V. An event v has two attributes: a type and an action (denoted type(v) and action(v), respectively). Two distinct events v_1 and v_2 are considered instances of the same type if and only if

type(v_1)=type(v_2). Two events of the same type will have the same associated action.

In general, an event is considered an atomic change in the global state of the system that the graph represents. For an event v, action(v) is the function that performs the state change. Some languages may partition the system into a set of entities, and confine an event to be a change in the local state of an entity, or on some restricted interface between entities. Depending on the language, an event may also involve a change in the graphical display associated with the system state.

For each event $v_0 \in V$, the set of contributing causes of v_0 is defined as

$$causes(v_0) = \{v|(v,v_0) \in V\}$$

The conjunction of all contributing causes of an event is the *cause* of the event. This allows us to create a *firing rule* for v_0:

$$\wedge\, causes(v_0) \rightarrow v_0$$

Thus, if $causes(v_0)=\{v_1,v_2,\ldots,v_n\}$, we have the firing rule $v_1 \wedge v_2 \wedge \cdots \wedge v_n \rightarrow v_0$, which states that v_0 occurs if $v_1,v_2,\ldots,$ and v_n all occur. Note that there is no notion of timing here other than the requirement that v_0 occur after $v_1,\ldots,v_n$.

Because of the presence of types, we can generalize a firing rule to apply to all events of a given type:

$$type(v_1) \wedge type(v_2) \wedge \cdots \wedge type(v_n) \rightarrow type(v_0)$$

According to the above rule, the occurrence of any events of the same type as $v_1,v_2,\ldots,$ and v_n will cause a new event of the same type as v_0.

This generalizing allows us to introduce disjunction into the model, since we may have two or more firing rules for events of a given type:

$$type(e_1) \wedge type(e_2) \wedge \cdots \wedge type(e_n) \rightarrow type(g_0)$$
$$type(f_1) \wedge type(f_2) \wedge \cdots \wedge type(f_n) \rightarrow type(g_0)$$

This may be simplified to

$$(type(e_1) \wedge type(e_2) \wedge \cdots \wedge type(e_n)) \vee (type(f_1) \wedge type(f_2) \wedge \cdots \wedge type(f_n)) \rightarrow type(g_0)$$

as the entire causality of events of type(g). In our execution model, however, we store the two rules separately.

The execution semantics represented by a causality graph are those denoted by its equivalent Petri net. For each causality graph C=(V,E), a Petri net graph $G=(V_G,A)$ (where $V_G=P \cup T$)* can be constructed by the following method:

1) For each event $v \in V$, add a node p_v to the set of places P.
2) For each event $v \in V$ such that $causes(v) \neq \emptyset$ (i.e., for every event with a non-empty cause), add a transition f_v to the set of transitions T representing the firing of event v.

---

* A Petri net[16] is formally a bipartite graph $G = (V_G,A)$ where $V_G=P \cup T$, P being the *places* in the graph and T being the *transitions*. P and T are disjoint. Each element of the set of arcs A either connects a place to a transition, or a transition to a place.

3) For each event $v \in V$, for each $g \in causes(v)$, add the edges (g,f_v) and (f_v,v) to the set of arcs A.

4) To execute the net, place a token on each place p_v such that $causes(v) = \varnothing$ (i.e., place a token on each place corresponding to a spontaneously firing event).

A sample transformation from causality graph to Petri net is given in figure 1. In the causality graph on the left, A and B are considered to cause C. In the equivalent Petri net at the right, transition f_C only fires when tokens reside in P_A and P_B. The result is to place a token on P_C.

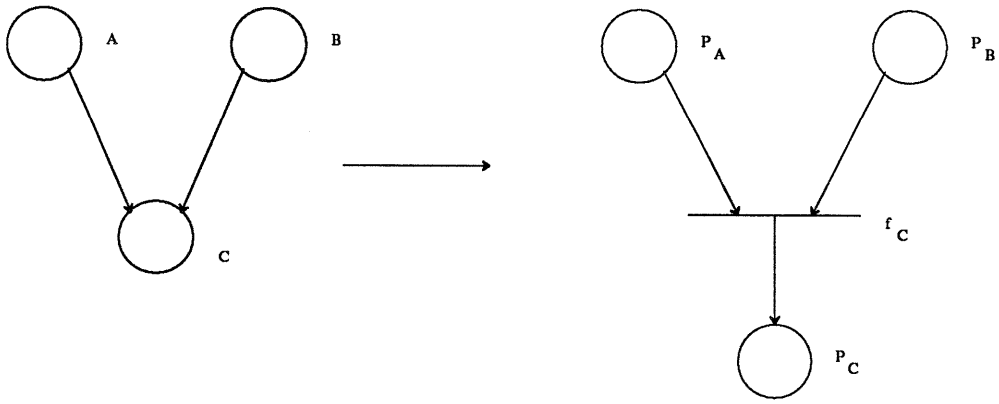


Figure 1. Transformation of a causality graph to an equivalent Petri net.

2.3. Programming as the construction of causality graphs

In a demonstration-based programming language, the user presents (that is, walks through) a scenario. Whenever the user draws or otherwise expresses a new event, that event is added to a causality graph that is constructed in parallel with the demonstration. The event may be defined graphically or textually, or by a combination of both graphics and text. As the event is defined, the user may (depending on the programming system being modeled) explicitly indicate the causes (for example, by pointing at them), or the system may deduce the causing events, or there may be some combination thereof.

This construction of the causality graph corresponds to the act of programming in an example/demonstration-based system (which will be referred to in the context of this paper as *programming* or *demonstration*), and the set of causality graphs generated during demonstration comprises the program itself (and will be referred to as the *program graph*, in order to distinguish it from *execution graphs*, to be discussed later).

During demonstration, the causality graph is constructed as follows. Beginning with an empty graph (where $V=E=\varnothing$), the user will repeatedly specify new events. The state changes associated with the event become the event's action, and the event's type is also specified. In some systems, the action is matched to previous actions to determine the new event's type; in other cases the user specifies the type explicitly (for example, through the selection of an icon).

After the event is specified, a new event node v_0, corresponding to the new event, is added to V. The causal events $v_1, v_2, \ldots, v_n \in V$ are either indicated explicitly by the user (for example, by pointing with the mouse to the visual analogues of the causal events, or textually), or they are computed using language-dependent heuristics, and the causal edges $(v_1,v_0),(v_2,v_0), \ldots, (v_n,v_0)$ are added to E. This process is repeated until the scenario is complete.

Several separate scenarios may be demonstrated in order to specify the program. Although each scenario is represented by a different graph, it better serves our purposes to think of the graphs from multiple scenarios as a single program graph composed of several unconnected subgraphs.

Once the program graphs have been constructed, the graphs are compiled into a rule base B of rules of the form

$$\text{etype}_1 \wedge \text{etype}_2 \wedge \cdots \text{etype}_n \rightarrow \text{etype}_0$$

where $\text{etype}_i = \text{type}(v_i)$ for some $v_i \in V$. In our model, this mirrors the process of compiling the demonstrated scenarios into an executable program.

The compilation process is simple. Given a causality graph G=(V,E), for each event $v_0 \in V$, with edges $(v_1,v_0),(v_2,v_0), \ldots, (v_n,v_0) \in E$, we add the rule

$$\text{type}(v_0) \wedge \text{type}(v_1) \wedge \cdots \wedge \text{type}(v_n) \rightarrow \text{type}(v_0)$$

to B. Where possible, duplicate rules are eliminated. In some systems with complex actions and event types determined directly from the action, this may not be possible, and we have no choice but to allow duplicates.

To execute the compiled program, the user sets up initial conditions in an *execution graph.* These may be expressed as events that set up the initial state. They may be either spontaneous events or events whose causes are derived from an empty initial event, although since rules for these events do not appear in the rule base, the choice makes no difference.

Following setup of the initial conditions, which establish a "seed" execution graph $G'=(V',E')$, an execution engine repeatedly matches rules in the rule base to the graph, adding nodes corresponding to the right hand side of a matched rule and executing the associated action. In order to avoid repeatedly applying the same rule in the same context, we require that no event be added to the graph if there is already another event in the graph of the same type and with exactly the same set of causes. The process of adding events and executing actions represents the execution of the program.

This we can describe an execution step as

$$\text{execution-step}((V',E') = (V' \cup \{v_0\}, E' \cup \{(v_1,v_0),(v_2,v_0), \ldots, (v_n,v_0)\})$$
such that there exists an $r \in B$ where $r = \text{etype}_1 \wedge \text{etype}_2 \wedge \cdots \wedge \text{etype}_n \rightarrow \text{etype}_0$
and $\text{type}(v_0) = \text{etype}_0$
and there exists a set $\{v_1,v_2, \ldots, v_n\} \subseteq V$
where $\text{type}(v_i) = \text{etype}_i$ for $1 \le i \le n$
and there does not exist $v \in V$ such that $\text{type}(v) = \text{etype}_0$ and $\text{causes}(v) = \{v_1,v_2, \ldots, v_n\}$

Program execution is modeled as a sequence of execution graphs $G_0, G_1, \ldots, G_n$, where G_0 is the seed graph, and $G_i = \text{execution-step}(G_{i-1}, B)$ (for $i \ge 1$). The final graph is

the least fix point, or the first G_i such that $G_i=G_{i+1}$. The effect of the program execution is the sequence of states derived from repeated application of event actions. Thus, if execution-step(G_i,B) adds a vertex v_i to the graph (i.e., yields a graph G_{i+1} with v_i), we get a sequence of program state $s_0,s_1,s_2, \cdots$, where S_i=action(v_{i-1})S_{i-1}, for $i \geq 0$.

It should be noted that execution-step is not a function since it is nondeterministic, and this mirrors problems of ambiguity or insufficient specification in demonstration-based languages. A set of events in the graph may trigger two or more new events (each through a different rule), or two different sets of events may each cause their own rule to fire. In addition, the same set of events may be added to the graph in different orders, possibly changing the effect of the applied actions. Some of this nondeterminism will be removed through the introduction of synchronization and timing in section 3, but much of it is inherent to demonstration-based languages and must be dealt with. In real demonstration-based systems, such problems are addressed through built-in heuristics or user intervention. Depending on the system being modeled, such disambiguating procedures must be built into the execution-step function.

This framework may also be used to model "live" programming environments, in which the system attempts to complete the scenario based on information that has already been entered. Examples of such systems are Eager[7] and Cara[6], where the systems attempt to complete actions where the current conditions match rules that have already been derived. Such live programming systems may be modeled by integrating all three stages of demonstration, compilation, and execution. During demonstration, while the program graph is being built, rules are compiled and added to the rule base as each event is added. At the same time, the system attempts to apply the rules already in the rule base to the incomplete graph. If the attempted application is incorrect, the new event must be removed and the action undone, and the rule base and/or the program graph must be altered so that the rule will not be incorrectly applied in the future. Any live programming system must provide facilities that reflect these changes in the underlying model.

It should be noted that, within the common structure of this execution model, a number of items are dependent on the particular language being developed. These include the definition of program state, the set of event types, the set of heuristics used to add causal edges to the graph in addition to the ones provided by the user as part of the event/cause stream, and the procedures used to resolve conflicts in the application of multiple heuristics during demonstration or multiple rules of behavior during execution. Such conflict resolution procedures may be arbitrary, involve user intervention, or use other schemes. All of these items may be considered parameters to the generalized model.

3. Adding timing and negation to causality graphs

3.1. Timing

For a demonstration-based system to be sufficiently powerful to be useful, a number of features must be present. First, it must be possible to order events and to refer to relative orderings of events as well as to the ages of events. Although not all demonstration-based systems require this facility, in many such systems the ordering implicit in the demonstration scenario is an integral part of the derived rules. Also, a scheme that allows for ordering and limits the number of events that can occur simultaneously

(generally to one event at a time) will avoid many of the ambiguity problems mentioned in section 2.

The second feature that must be accounted for is negation. While the basic model allows the user to specify that an event occurs because a previous event occurred, it should also be possible to specify that an event occurs because a previous event did not occur. Thus, a user of the MFD system[5] might wish to draw a scenario indicating that a message is resent when an acknowledgement is *not* received within a certain time. Such scenarios are very difficult to specify entirely through graphics: demonstration scenarios illustrate *what happened*, not what *didn't* happen. Such negative causalities must be illustrated at least partly with text. (In fact, the original motivation for this work was a search for suitable graphical representations of negative causality. That research is ongoing.)

We make two assumptions in our model. While they are limiting, we believe that they are useful for a large class of demonstration-based systems. First, only one event may be executed at a time. As we shall see in the example in the next section, this can be generalized to multiple parallel entities each executing one element at a time.

The second assumption is that the internal clock, and therefore the timing scheme, is event driven. Each time an event fires, the clock ticks. Relative timings (for example, ''three clock ticks ago'') refer to intervening event firings. If no rules may fire, the system will halt, unless the programmer includes rules with null actions whose sole purpose is to force clock ticks, and consequently force the passage of time. This scheme will not allow references to absolute time units (for example, ''three seconds ago,'' or ''twenty-five minutes after event A fires''); accommodating such time references will require further research. Use of token aging[18] is a possibility. The issue, however, does not appear to be a significant one in most demonstrational systems.

In order to enforce timing, we must add synchronization edges to the causal graph. Otherwise, there is no way to enforce the fact that, for example, event B only happens if event A happened three events previously. If we merely say A→B, B could occur immediately after A. To accommodate a rule like A(−3)→B, we need to know that A not only happened in the past, but that it occurred three events ago, and that requires an explicit notion of synchronization.

In order to permit timing, we add a set of synchronization edges S to the graph, so that a causal graph G is now the triple (V,E,S). An edge $(v_1,v_2)\in S$ if and only if v_2 is the event immediately following v_1, in the sense that no intervening events take place. Note that the synchronization edges do not imply any causality, but only sequentiality *within the scenario.*

We also require that the synchronization edges form a single chain, so that $(v,u),(v,w)\in S$ implies u=w, and likewise $(u,v),(w,v)\in S$ also implies u=w.

With both causal and synchronization edges, we can make assertions about timing. For example, figure 2 shows a causality graph that may be interpreted as specifying that event B depends on event A occurring immediately previously, while event C depends on both B occurring immediately previously and event A occurring two events previously.

When causality graphs are translated into Petri nets, synchronization edges are handled in the same way as other edges. They are not translated into the rule base, although we shall see that timing information derived from synchronization chains is attached to

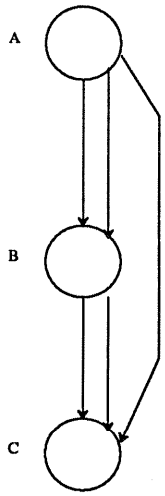


Figure 2. Causality graph with synchronization edges.

causal edges.

When constructing the program graph, the user refers to previous events as causes of the new event. Timing information is associated with the causal edge either explicitly (in that the user indicates the timing considerations that may hold) or through heuristics (that is, the system assigns them). One possible heuristic is that the number of intervening events between the new event and the causal event in the program graph must be exactly preserved in all execution graphs based on the derived rules.

Timing information may be constant (where -1 refers to the previous event, -2 to the event before that, and so forth) or inequalities (for example, ≤ means "at least two clock cycles ago," ≤2, ≥5 means "between two and five clock cycles ago"). These timing conditions are attached as labels on the causal edges, and upon compilation are incorporated into the left side (guard) of the firing rules:

$$\text{etype}_1(-2) \wedge \text{etype}_2(\leq -1, \geq -5) \rightarrow \text{etype}_0$$

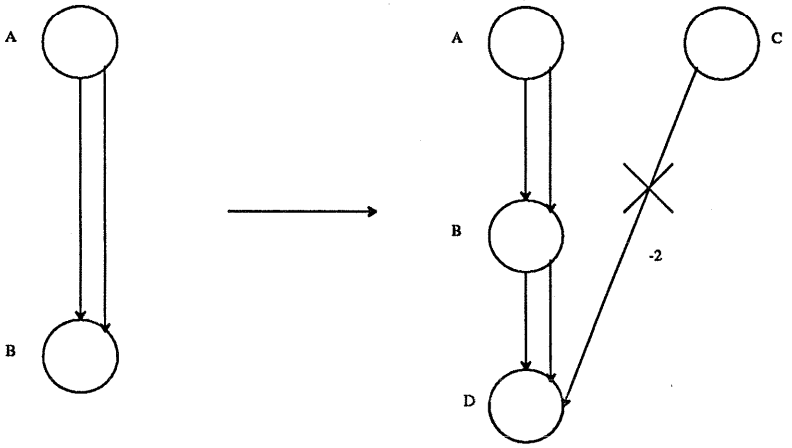
A guard term etype is shorthand for etype(-1).

The timing portions of the conditions are used by the execution engine to add new events to the execution graph. The engine not only attempts to match event types in rules to those in the graph, but also ascertains that the timing conditions are obeyed. For each matched event type, the execution engine searches back from the end of the synchronization chain to the matched event and attempts to match timing conditions. If they match, the new event v_0 is added to the set V', causal edges (labeled) are added to E', and v_0 is connected to the end of the synchronization chain. (v_e is the end of the chain if and only if $(u,v_e) \in S$ for some u, and there exists no $w \in E$ such that $(v_e,w) \in S$. (v_e,v_0) is then added to S.)

As before, when an event is added to the execution graph, its action is executed.

3.2. Negative causality

In order to accommodate situations where an event occurred because a previous event did *not* occur, negative causality is incorporated into the model. Causal edges in program and execution graphs may be annotated with a negation symbol. This is an indication that a guard term must not be satisfied if the rule is to be satisfied; that is, that the causal event may not be found by the execution engine within the execution graph for the rule to hold. Negated edges may also be annotated with timing conditions to control the execution engine's search through the graph for a matching. Since the causal event does not appear in the execution graph, the execution engine adds it to the graph (although no action is performed) and a negated causal edge is also added. Figure 3 shows the adding of an edge based on negative causality.



Matching rule: $B(-1) \wedge \overline{C(-2)} \rightarrow D$

Figure 3. Matching using negative causality.

In order to translate graphs with negative causality to equivalent Petri nets, we must first add to the causality graph a synchronization edge that places the negative event on the synchronization chain at some point conforming to the timing considerations. As long as these conditions are satisfied, the exact place on the chain does not matter. (See figure 4).

The resulting graph is converted to a Petri net as previously described, with the negation edge handled in one of two possible ways. The first is to use an inhibitor edge in place of the the negative causality edge. If a transaction node has an inhibitor edge incident on it, it may only fire if there is no token on the place from which the inhibitor edge originates (provided, of course, that all other firing conditions are satisfied). If (B,C) is a negative causality edge, then the Petri net will have an inhibitor edge from p_B to f_C. (See figure 5.)

The second method is to use a switch[1]. A switch is a special node with two input ports, Synch and Data, and two output ports, F (Full) and E (Empty). If tokens are waiting at both the Synch and Data ports, they are consumed, and a token is output on the F port. If there is a token on the Synch port but no tokens on the Data port, the Synch token is consumed and a token is output on the E port. Otherwise the switch does

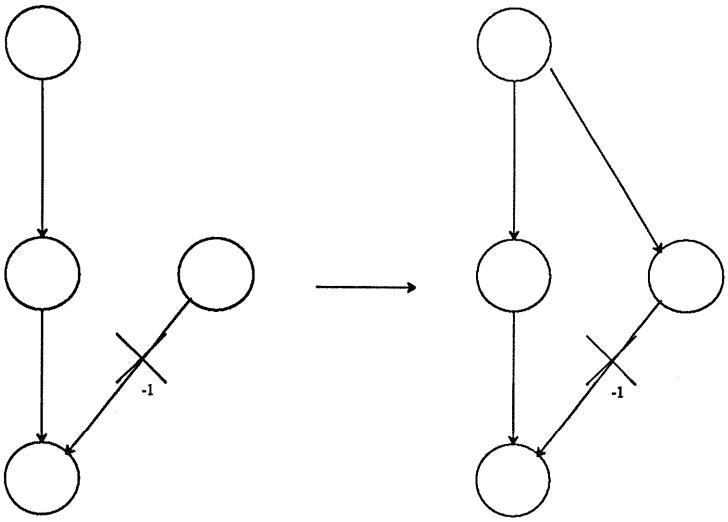


Figure 4. Placing a negative event on a synchronization chain.

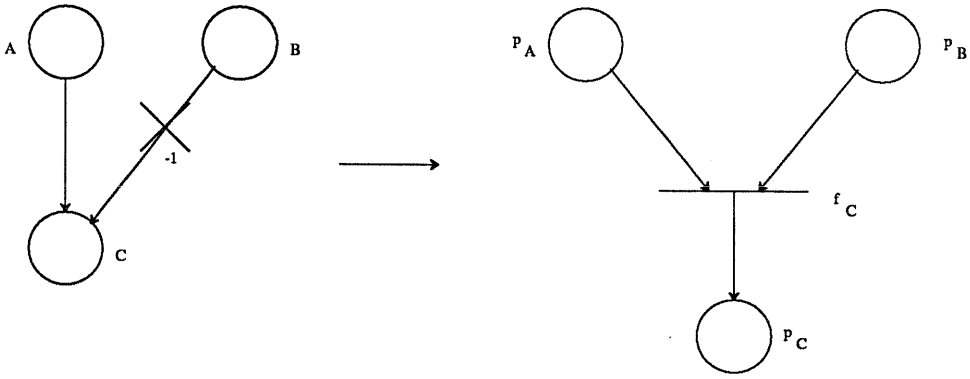


Figure 5. Translating a negative causality edge to an inhibitor edge.

nothing. If (B,C) is a negative edge, the translated Petri net will have an edge from p_B to the data port of the switch, and there will be an edge from the E (empty) output port of the switch to f_C. (The F/full port is simply grounded.) To synchronize the switch, its synchronization port is connected to the output synchronization edge from the negative event's sibling on the synchronization chain. This guarantees that the switch is only enabled when the event that happens instead of B (A, in figure 6) fires.

4. An example

The following example is derived from MFD[5], a demonstration-based language for the specification of communications protocols. The primary events are message receipt and transmission, and the language supports multiple communicating units (each of which can independently and simultaneously fire events).

The protocol we wish to define will, upon receipt of a message A, transmit a message B. When B is received, C is transmitted back. When C is received *and* the sender

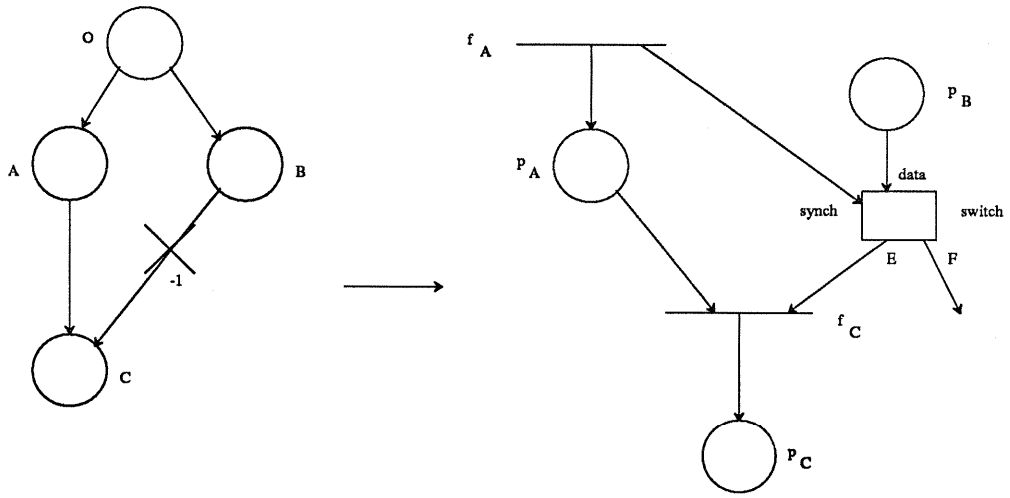
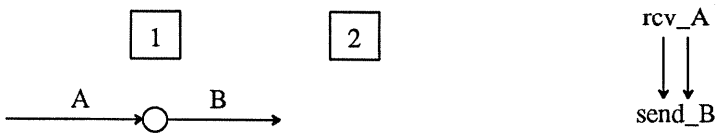


Figure 6. Translation of negative causality to Petri nets using switches.

has previously sent a message B, it sends a message D.

We assume that there are two units participating in the conversation, and that they are independently timed. We present the step-by-step specification of the protocol, and, in parallel with that, the construction of the program graph. Note that the event types in the program graph are the receipt of A (rcv_A), the transmission of B (send_B), rcv_B, send_C, rcv_C, and send_D.

The first step (figure 7a) is to show the receipt of the initial message A, and the resulting transmission of B. In the program graph, we show that send_B is caused by rcv_A (figure 7b). Note the presence of the synchronization edge in the program graph due to the fact that the events occur in the same communicating unit.



(a) (b)

Figure 7. First step in defining the protocol.

a) MFD specification

b) Program graph.

In the second step (figure 8a), unit 2 receives the message B and returns a message C. Since 1 and 2 have separate clocks, there is no synchronization edge between send_B and rcv_B (figure 8b). In the third and final step (figure 9a), the receipt of C and the previous transmission of B cause the transmission of D. In previous steps, transmission of a message immediately after a receipt implied that the receipt was a cause of the transmission. In this case, since there is a previous cause that is not incident upon the transmission event, we must indicate the connection through a *history edge* which is annotated to

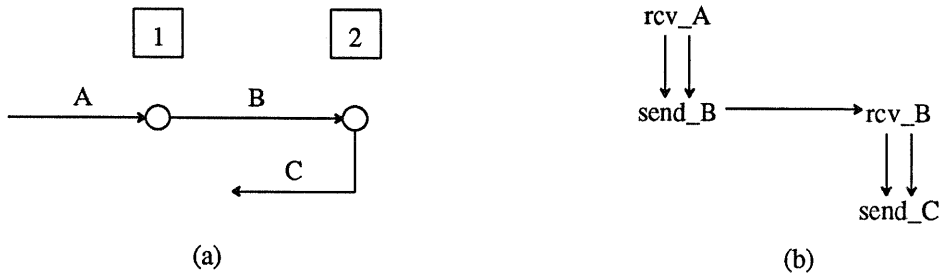


Figure 8. Second step in defining the protocol.

show that this event may have occurred at any time in the past. Corresponding to this history edge is an additional causality edge from send_B to send_D (figure 9b). Also note that we have to place a synchronization edge between send_B and rcv_C since the corresponding events must occur in that order.

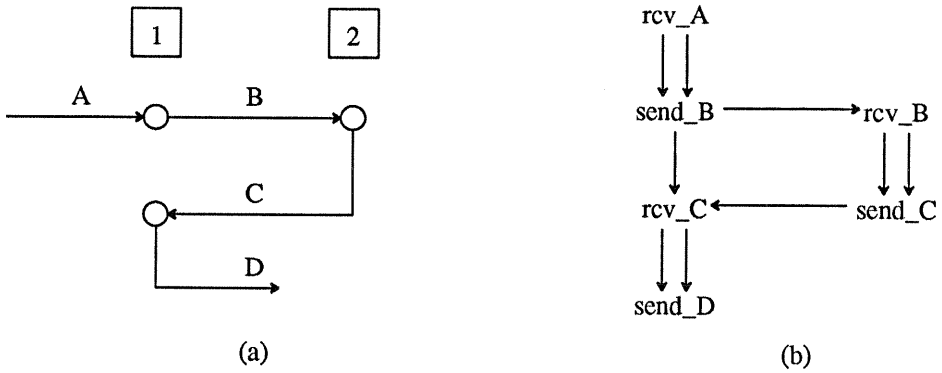


Figure 9. Third step in defining the protocol.

We see the following causal relationships in the final program graph:

- send_B was caused by rec_A.
- send_C was caused by rcv_B.
- send_D was caused by rcv_C and a previous send_B.

In addition, rcv_B depends on send_B, and rcv_C depends on send_C. This is a property of message transmission in that a message's receipt depends on its transmission. Such edges only occur between events of different units.

The following rules are derived from the program graph:

$rcv\_A \rightarrow send\_B$
$rcv\_B \rightarrow send\_C$
$send\_B(\leq -2) \wedge rcv\_C \rightarrow send\_D$
$send\_B \rightarrow rcv\_B$
$send\_C \rightarrow rcv\_C$

Note that, although the MFD program specifies that the transmission of B that causes the transmission of D may occur in the immediately previous event, since the

program graph representation considers the receipt of C and the transmission of D to be separate events, the causal edge corresponding to the history edge must be annotated "≤-2".

Our specification allows two simultaneous conversations to take place - one in each direction. For example, if we set up the initial events rcv_A, one for each unit, we would build an execution graph reflecting the two simultaneous conversations (figures 10 and 11).

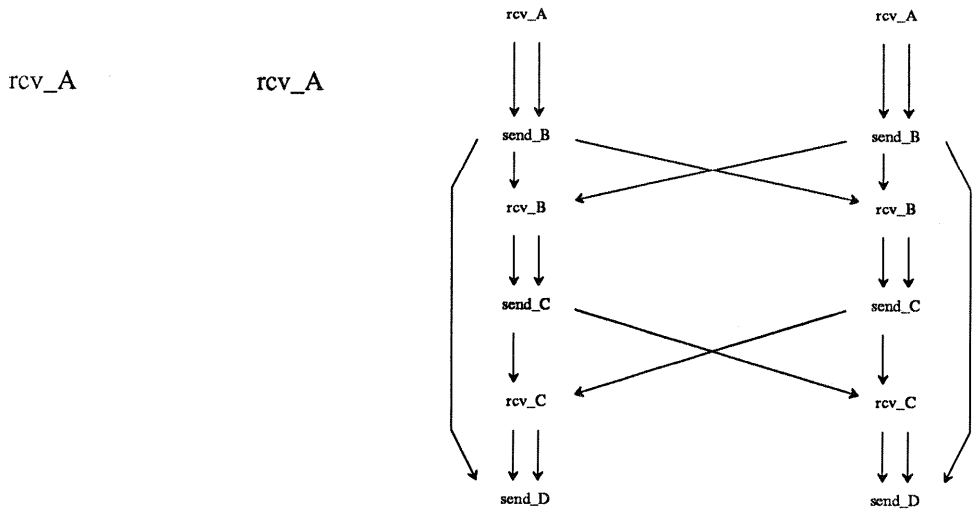


Figure 10. Execution graph for concurrent conversations

a) "seed" execution graph.

b) Completed execution graph.

Similar graphs can be built up for other, more complex, diagrams, as well as for diagrams derived from other demonstration-based systems.

5. Discussion

The execution model for demonstration-based programming languages that we have presented in this paper is sufficiently general to cover many such systems. Having such a model confers a number of benefits:

- The model will allow us to develop a theory of demonstration-based languages, in which we can analyze problems of ambiguity, timing, and negation.
- The model provides a formal method for defining the semantics of a given system.
- Because the model includes an execution engine, it is itself executable. Appropriate tools can be built to allow the rapid design and implementation of prototypes for demonstration-based systems.

One other potential benefit of the model is that it offers the possibility of exploring a potentially very powerful debugging technique for certain types of visual languages. When the output of the program is in the same form as the program itself (a reasonable

expectation in a demonstration-based language), it should be possible in some cases to take erroneous output from a buggy program, edit it to reflect the correct behavior, and recompile the output as a valid program. The now-correct rules are then extracted and added to the rule base. Research remains to be done on how to identify the incorrect rules that must be replaced in this process. Such a debugging scheme has been proposed under the heading of Visualization-Based Visual Programming[3], and has been suggested by Kurlander and Feiner under the heading of Editable Graphical Histories[11]. The execution model presented here supports this form of debugging by offering program graphs and execution graphs in the same format. The two graphs reflect the same rule base, and the execution graph can be compiled using the same process as the program graph. Consequently, a model of visualization-based debugging could be developed based on this model.

The execution model suggests a rule-based approach to demonstrational language implementation. This is exactly the way Cara and MFD were implemented. In these systems, the programmer employs a graphical front end to specify events in the scenario. Cara employs heuristics to find the causal relationships, while MFD requires that the programmer to specify them explicitly. Using this information, the compiler creates rules in a rule base. A simulator then executes the rules to repeatedly alter the state, in effect creating an execution graph.

There are a number of advantages to such a rule-based approach to implementation. The rules in a rule base may be independently edited, or even deleted, without disturbing the function of the other rules. Multiple applicable rule firings indicate ambiguities in the specification. Both these properties may be more difficult with a conventional state machine representation. Of course, state machines are more efficient than rule-based systems. This problem may be solved by converting the rule base to an extended finite state machine once the system has been specified and debugged, although the process is not trivial.

The execution model also suggests a number of interesting aspects of demonstrational language design, particularly in the area of negative causality. As mentioned earlier, the original motivation of the current work was to find suitable graphical representations for negative causality. It did not seem advisable to design a demonstrational system that required the negative event to be shown in the demonstrated scenario. This would be difficult if we wished to confine ourselves to specifying only those events that do occur, and it also seemed to introduce problems in the readability of programs, since it was believed that a user would assume that every event shown on the screen was a positive event. The rule-based model we propose, however, requires indication of the negative events, both in the generated rules, and in the causal graphs. Since all the information in the causal graph is supplied through the demonstrations, and the negative events in the graph contain the same type and timing information as positive events and are only distinguished by their negative causal edges, it seems necessary to enter the negative events in the same manner as the positive ones. The only alternative seems to be to enter them textually, but we hope to use that solution only as a last resort. In the meantime we propose to investigate graphical representations for negative events that are distinguished from positive events in a way that is both readable and semantically sound.

6. Future work

We are pursuing a number of directions suggested by the work described in this paper. We are developing a formalization of the model described here. This involves describing the domains on which the model acts, and describing the three stages of the model (demonstration, compilation, and execution) as functions acting on those domains[4].

We are also currently working to describe a number of demonstrational systems (including MFD) using this model. This involves finding definitions for the various model parameters mentioned earlier in the paper, including state, event types, and heuristic and rule conflict resolution. In the case of MFD, this will be the actual formal specification of the semantics of the language. In the case of the other languages, these definitions will serve to explore the power and usefulness of the model, and will provide a means for comparing the various languages on a number of levels.

Finally, we hope to use the model to help develop a taxonomy of demonstrational languages. This would allow us to formally characterize the distinctions between scripting systems, programming-by-example, and programming-with-example, and possibly identify additional paradigms.

In the long run, it is possible that such work might serve as to guide implementation strategies and architectures for demonstrational systems, and possibly lead to the automatic implementation of such systems from formal specifications.

Acknowledgements

The author would like to thank Ellen Phelps, who contributed to an earlier version of this paper.

This work was partially supported by the National Science Foundation under grant no. CCR-9208486.

References

1. J. Baer and C. Ellis, "Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 6, pp. 394-405, November 1977.
2. S.-K. Chang, M. J. Tauber, M. Yu, and J.-S. Yu, "A Visual Language Compiler," *IEEE Transactions on Software Engineering*, vol. 15, pp. 506-525, May 1989.
3. W. Citrin, *Visualization-Based Visual Programming,* Dept. of Computer Science, University of Colorado, Boulder, July 1991. Technical Report CU-CS-535-91
4. W. Citrin, *Towards a Formal Model for the Specification of Demonstrational Languages.* (In preparation)
5. W. Citrin, "Design Considerations for a Visual Language for Communications Architecture Specifications," in *Proceedings 1991 IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.
6. A.A.R. Cockburn, W. Citrin, R.F.Hauser, and J. von Kaenel, "An Environment for Interactive Design of Communications Architectures," *Proc. 10th Intl. Symposium on Protocol Specification, Testing, and Verification*, Ottawa, June 1990.

7. A. Cypher, "EAGER: Programming Repetitive Tasks by Example," *CHI '91 Conference Proceedings*, pp. 33-40, New Orleans, April 27 - May 2, 1991.

8. W. Finzer and L. Gould, "Programming by Rehearsal," *Byte*, vol. 9, no. 6, pp. 187-210, June 1984.

9. E. P. Glinert and J. Gonczarowski, "A (Formal) Model for (Iconic) Programming Environments," in *Human-Computer Interaction -- INTERACT '87*, ed. B. Shakel, Elsevier Science Publishers, Amsterdam, 1987.

10. D. Halbert, *Programming by Example,* Computer Science Division, University of California, Berkeley, CA, 1984. PhD Thesis

11. D. Kurlander and S. Feiner, "Editable Graphical Histories," *Proc. 1988 IEEE Workshop on Visual Languages*, pp. 127-134, Pittsburgh, PA, October 1988.

12. D.L. Maulsby, I.H. Witten, and K.A. Kittlitz, "Metamouse: Specifying Graphical Procedures by Example," *Proceedings SIGGRAPH '89*, Boston, July 1989.

13. B. A. Myers, "Creating Interaction Techniques by Demonstration," *IEEE Computer Graphics and Applications*, pp. 51-60, September 1987.

14. B.A. Myers, "Taxonomies of Visual Programming and Program Visualization," *Journal of Visual Languages and Computing*, vol. 1, pp. 77-95, 1990.

15. G. J. Nutt, *A Formal Model for Interactive Simulation Systems,* Dept. of Computer Science, University of Colorado, Boulder, September 1988. Technical report CU-CS-410-88

16. J. L. Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, New Jersey, 1981.

17. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, pp. 57-69, August 1983.

18. P. D. Stotts, "The PFG Language: Visual Programming for Comcurrent Computation," in *Visual Programming Environments: Paradigms and Systems*, ed. E. P. Glinert, pp. 558-565, IEEE Computer Society Press, Los Alamitos, CA, 1990.

19. G. Tortora and P. Leoncini, "A Model for the Specification and Interpretation of Visual Languages," *Proc. 1988 IEEE Workshop on Visual Languages*, pp. 52-60, Pittsburgh, PA, October 1988.