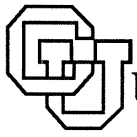
Design Considerations for a Visual Language for Communications Protocol Specifications

Wayne Citrin

CU-CS-536-91 July 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Design Considerations for a Visual Language for Communications Protocol Specifications

Wayne Citrin

CU-CS-536-91

July 1991

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

Design Considerations for a Visual Language for Communications Protocol Specifications

Wayne Citrin

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

citrin@soglio.colorado.edu

ABSTRACT

Communications protocol specification is a complex problem domain in which a visual paradigm may be employed. This paper describes the desirable characteristics of a visual specification language for this problem domain, and presents a preliminary design for such a language, based on message-flow diagrams.

1. Introduction

The development, over the past 5-10 years, of visual programming has opened up a new field of language design. Visual languages are considered advantageous for a number of reasons: they permit the possibility of programming in a form representing the designers' mental models of the problem being solved; they reduce or eliminate complex syntax, and may reduce the problems of naming; they provide the programmer with greater insight into program structure and function; and they may allow the programmer to observe the effects of partial, incomplete programs, thereby assisting development. Combined with the related field of program visualization, they have the potential for increasing programmer productivity and for introducing many novices to the programming process.

Many of the early visual languages were designed to prove the feasibility of the visual paradigm, or to introduce novice programmers to computers[15]. Such languages often had limited power or limited ability to scale up to large problems. While a number of languages and systems have been developed to solve problems in various domain-specific areas such as databases[23], user interface design[19], or spreadsheet calculations[3], there is still a need for languages that are suitable for use by experts (in that they assume extensive knowledge of programming, or the application area, or both), and that scale up to large problems. In this way, we can apply the power of the visual model to large, specialized programming problems.

One such complex problem domain is the specification of communications protocols. This is the domain to which we have turned our attention. After briefly explaining the problem domain, this paper will review visual models that have already been attempted, outline the design criteria for a language to specify communications protocols, and present a preliminary design for such a language.

2. Problem domain - communications protocol specification

The problem of specifying communications protocols has long occupied that branch of computer science concerned with computer communications and networks[22]. The designers of communications protocols must deal with many issues: assuring that information is not lost (or that it is retransmitted if it is); meeting performance criteria for speed, bandwidth, or capacity; dealing with multiple simultaneous conversations; handling routing and flow control; enforcing security rules; and dealing with devices and transmission media of many differing characteristics, among other problems. To handle these issues, behaviors, known as protocols, are designed. These protocols consist of a set of symbols or messages that may be exchanged between communicating entities in the networks, as well as rules concerning the symbols' meaning, and the order in which they may be exchanged. Protocols may also contain assumptions about the characteristics of the elements of the network. Since real-life protocols are large and complex, they are usually structured into a number of levels, each of which handles a particular function and which employs the facilities of the levels below. For example, one level may handle entire transactions, another level may divide messages into packets, a third level may handle encryption and authentication, and still another level may deal with the physical properties of the communications medium. The structuring of protocols into multiple levels yields *communications architectures*. A detailed discussion of communications architectures is beyond the scope of this paper.

The design of a communications architecture is usually given as a formal specification which is used as a guide for implementers[12]. The specification outlines the allowable behaviors, which the implementer is free to implement in any way he or she chooses. (Two protocols conforming to the same specification but with different implementations should still be able to communicate with each other.) Some specification methods yield executable specifications, that is, a runnable, albeit possibly inefficient, implementation of the protocol. Two formal specification methods, Estelle[2], based on extended finite-state machines, and LOTOS[6], based on process algebra, are international standards. Both have, or are in the process of having created, a visual representation, which will be will be discussed in the next section.

In addition to being able to formally specify a finished protocol, it would be useful if an executable specification method could simulate partly completed specifications, or a single level of a specification, where the lower layers are not yet specified. It would also be advantageous if the specification method and its simulator allowed easy modification and resimulation of the changed protocol, thereby allowing exploratory simulation of various alternatives. Rule-based specification methods[8, 9] provide these abilities.

The process of protocol simulation yields itself naturally to visualization. For example, traces of a simulation may be collected and displayed as message-flow diagrams (see below). Such displays are substantially easier to read and understand than textual traces, and can greatly assist in understanding and debugging protocols under development.

In addition to visualization of simulations, the act of protocol design itself lends itself to a visual treatment. Communications protocols are concurrent systems, and, like other kinds of concurrent systems[21], lend themselves to visual programming. In addition, the issue of timing, which is a distinctive aspect of the behavior of communicating systems, may be usefully illustrated through a visual dimension. The fact that protocol specifications are often accompanied by graphical documentation([1], for example)

is highly suggestive of the important role that visual specification can play. We predict, although we do not yet have any experimental evidence, that a usable visual language for communications architecture specification will improve designer productivity, reduce errors, and decrease the time needed to debug those errors that do occur.

3. Previous efforts

GROPE (Graphical Representations Of Protocols in Estelle)[20], is a visualization environment for communications architectures specified in Estelle. Since it is based on extended finite-state machines, it employs the very natural visual representation of finite-state machines as directed graphs whose nodes represent machine states, and whose edges represent transitions. GROPE reads in a textual Estelle specification, parses it, and translates it into graphical form. The protocols are shown in the form of the abovementioned graphs, and the structure levels are represented graphically by multiple layers of boxes with communications queues between them. When GROPE is used to simulate a specification, the active states and transitions are highlighted, and the message traffic between layers is indicated by message items on the communications queues. Although GROPE is a visualization environment and not a programming environment, it should be fairly straightforward to add whatever is required to make it a language. Figure 1 shows an example of a specification visualized using GROPE.

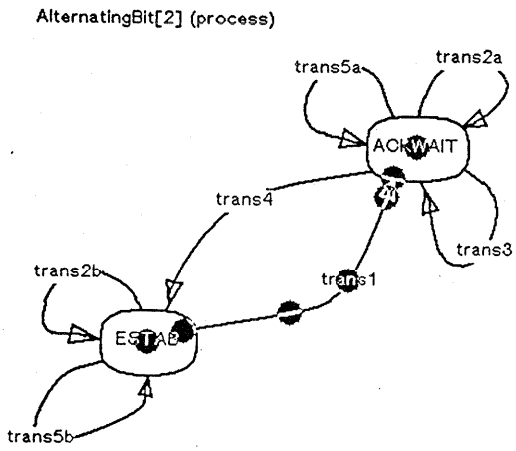


Figure 1 - a GROPE visualization (from[20])

LOTOS, as mentioned earlier, is based on process algebra, and has no generally accepted visualization. Two visual versions of LOTOS are being prepared: G-LOTOS[5], and UO-GLOTOS[7]. Both of these schemes are based on direct mappings from textual LOTOS syntax to the visual representation. While both versions are intended to help a reader understand the complex structure of textual LOTOS specifications, one of the main objectives of G-LOTOS is to come up with a visual representation that has the same formal properties as the underlying textual specification. Visually suggestive indications of the behavior or operation of the protocol was not an issue. In UO-GLOTOS, such suggestions of behavior were a more important priority. Figures 2a and 2b provide some examples of typical G-LOTOS and UO-

GLOTOS code, respectively.

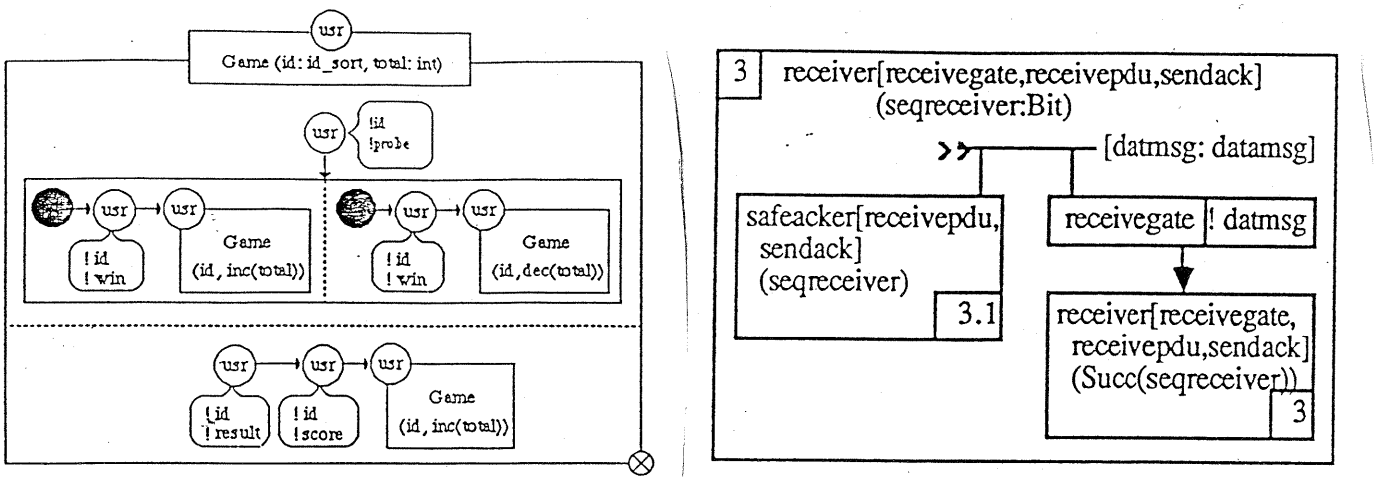


Figure 2a - G-LOTOS specification (from[5])
Figure 2b - UO-GLOTOS specification (from[7])

Another proposed visual representation is the extended sequence charts (ESC's) associated with the SDL language[17]. As described, however, the ESC's appear to be a documentation technique rather than a visual specification method. ESC's illustrate a typical dialogue and augment it with notations describing the state of each communicating entity. They appear to be a hybrid of message-flow diagrams (see below) and finite-state machines. There is no indication, however, of whether a particular action only occurs in the context in which it appears, or if it may be generalized. There is also no indication of whether or not data may be stored in any form other than as a discrete state. Such an approach does not have the generality that we desire, although it is higher in level, and more behavior-oriented, than the graphical LOTOS representations. Figure 3 shows an example of an ESC.

Cara[11] is a visual specification environment employing message-flow diagrams as the visual representation. The diagrams used, however, are ambiguous, and therefore the user must interact extensively with the designer in order to arrive at an unambiguous interpretation. The specification procedure in Cara is as follows: the user/designer draws a message-flow diagram on a specially supplied graphical editor. Every time an event is specified, the system employs a set of heuristics in order to derive a textual rule describing the action, and presents that rule to the user. The user is free to edit the rule, and the completed rule is returned to the system, where it is deposited in a database. Later, if the user specifies another action, and the condition fits an already specified rule, the system will use that rule to complete the visual representation of the action on the diagram.

Another feature of Cara is the ability to simulate the specified protocols. The user sets up a configuration and some initial conditions, and the simulation is displayed graphically in the form of message-flow diagrams. These visualized traces may be compared to the original graphical specifications to see if they have performed as expected.

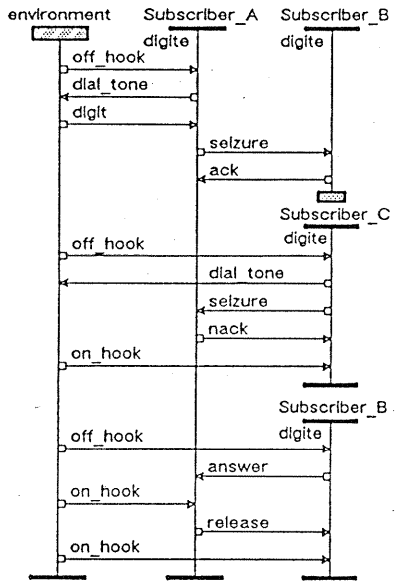


Figure 3 - an Extended Sequence Chart (from[17])

We will discuss message-flow diagrams in a later section, but here it will suffice to say that they contain many features that are desirable for our purposes. The work described in this paper is in large part based on the Cara project and its visual representations.

4. Design criteria

In this section, we will briefly discuss a number of desirable features of a visual language for communications protocol specifications.

The language should be visualization-based. A good place to start when looking for a visual programming model is to look for models used in visualizations. Such visualizations are found in program visualization systems, but also, perhaps more importantly, in documentation. Since visualization models need not be chosen for their executable semantics (unlike visual programming models), it is a reasonable assumption that they are chosen because they reflect a natural mental representation of the problem being solved, and therefore it should be far simpler to program a solution in one of these models than it would into a lower-level model in which executable semantics are the primary consideration. If the model we have chosen does not have an executable semantics, we can augment the model so that it has one.

A second advantage to using a visualization-based model is that if visualizations are in the same form as programs, an incorrect program can be debugged by correcting the behavior on the erroneous visualization (by editing the visualization) and recompiling the corrected visualization as a program. If this can be accomplished, it would be a very powerful debugging technique. We will have more to say about this in the context of our language later in the paper.

The language should be static. In a number of visual languages, particularly those that are demonstration-based[14, 19], the actual visual image alone is not the program; the program also includes the gestures and manipulations - it is *dynamic*. In our language, we wish the picture, or collection of pictures, to be the complete and executable program. This is desirable because the visual protocol specifications do not function simply as a program; they also function as documentation, that is, as a means

for one person to communicate ideas to another. As documentation, the specifications should be suitable for inclusion in books, manuals, and reports. Hence, they must function as static pictures.

Another reason for the desirability of this static property is that the form of the picture (that is, its appearance), rather than the way in which it is assembled, should contain the picture's meaning. (Such a class of visual programs was first described by Kahn and Saraswat in their work on *complete visual programming*[18].) Thus, the diagrams ought to be able to be assembled in any direction: from the top of the diagram to the bottom, from the bottom to the top, or from the inside out. This allows a more natural style of drawing, and also eliminates the need for a specialized language editor (although one can be provided). Instead, general graphics editors, or even pencil and paper (where the drawing is optically scanned) should be possible.

Related to the previous criterion, *the visual representation should be suitable for various media.* Certain features, like pop-up windows and menus, are suitable for bit-mapped displays, but are not so suitable for versions of the program printed on paper. Likewise, color may be difficult to reproduce on paper. There should be either a single form of the representation that is suitable for video screen or paper, or there should be comparable forms, easy to switch between, for both paper and screen.

The language should use text only for those things that are cumbersome to describe visually. Certain actions, such as message exchange, are easily visualized, and should have visual counterparts in our language. Other aspects, such as operations on numbers, may have no natural visual equivalent. If that is the case, it is acceptable to use a textual notation, but the interface between the visual elements of the language and the textual ones should be simple and natural.

The language should scale up to large problems. Many experimental visual languages do not scale; often this was not an objective in their design. In our problem domain, the specification of communications architectures, real-life examples can be quite large. We estimate that even a moderate-sized architecture could require several hundred diagrams to describe. Our language needs facilities for modularization and abstraction.

The final two criteria are more specific to our problem domain. The first is that *the language should allow early, high-level simulation.* This is a consequence of the type of development environment and methodology we wish to promote. Users should be encouraged to experiment with their designs early on, so that mistakes may be discovered and corrected at an early stage. This was also a prime motivation of the Cara project[11].

The second is most important: *the language should provide a complete, unambiguous specification of the protocol*, at least, when the textual elements are included. This differs from methods such as extended sequence charts and message-flow diagrams as employed in Cara in that these methods yield ambiguous specifications that require further input from the user to make them unambiguous and executable.

5. Message-flow diagrams

Message-flow diagrams, the visual model we have chosen for our language, are an almost ubiquitous representation, appearing in varying formats in protocol documentation[1] and textbooks[22]. The diagrams represent, graphically, a trace of the output, or message-exchange, behavior of the protocol. The

entities participating in the protocol (known as PEs, or "protocol entities") are represented by a row of boxes along the top of the diagram. Actions performed by each entity are shown in the column of space in the diagram below the entity's box.

Messages exchanged between PEs are represented as arrows leading from sender to receiver. The arrows are labeled with the text of the message, some of which may be left as variables. Where a PE sends or receives a message, a small circle, denoting an *event point*, is drawn. This event point is treated as an instantaneous and atomic action encompassing all associated message receipts and transmissions, as well as any other actions performed by the PE at that time that may not be visible in the diagram. Time is assumed to flow downwards within a PEs column, or along message transmission arrows. The former condition ensures that an event point occurs before any of the PE's other event points that appear below it, and the latter condition ensures that the event that caused the transmission of a message occurs before the event that receives the message. These relationships result in a partial order on event points.

Each message-flow diagram describes a sample conversation, or scenario, in conformance with the protocol. A set of such diagrams outlines the limits of acceptable behavior for a protocol.

Figure 4 provides a sample message-flow diagram.

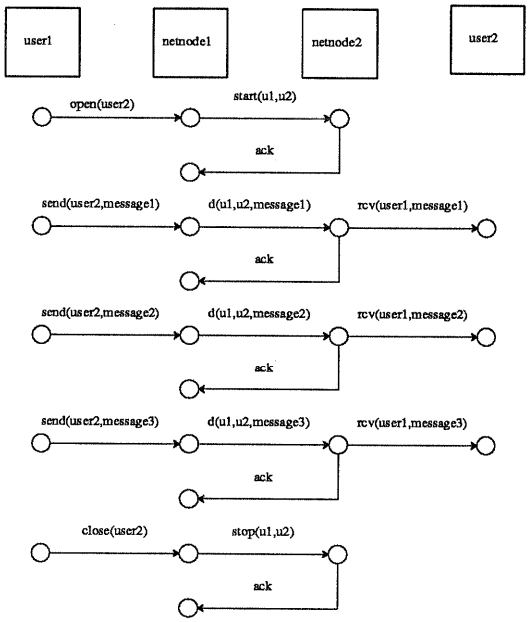


Figure 4 - a message-flow diagram

The diagram shown in figure 4 illustrates a sample dialogue between two entities, user1 and user2, mediated by two other entities, netnode1 and netnode2. At the beginning of the conversation, user1 indicates that it wishes to open a conversation with user2 by sending an **open** message to its agent, netnode1. netnode1 then notifies user2's agent, netnode2, of user1's desire to start a conversation and netnode2 acknowledges this request. user1 then begins to send messages to user2. These are repackaged and relayed by netnode1 to netnode2, which acknowledges each message and notifies user2 of the receipt of a message from user1. Finally, user1 closes the conversation, netnode1 notifies netnode2 of this, and netnode2 acknowledges it.

As a specification for the protocol, however, the diagram has numerous ambiguities and raises numerous questions. Is it possible for user2 to initiate a conversation with user1, and does the protocol behave in the same way if that happens? Is it necessary for an **open** command to be sent before any data is sent? What happens if any data is sent before the conversation is formally opened? What happens if data is sent after the conversation is closed? What happens if a message is not acknowledged? Does the diagram indicate that exactly three pieces of data may be sent, or do those **send** messages indicate that any arbitrary number of messages may be sent, or is there an entirely different interpretation? In order to assign executable semantics to the diagram, these ambiguities must be resolved. The approach we have chosen, which is described in the next section, is to annotate the diagram with further graphics and text in order to indicate *why* each event point occurs when it does. Such an event could then occur any time those conditions hold.

Nevertheless, the use of message-flow diagrams as a basis for a visual language for communications architecture specifications has several advantages. It expresses the message-passing behavior of the protocol, which of course is the purpose of the protocol (i.e., to exchange messages). This differs from finite-state machine-based approaches expressing the state transitions, which are only an artifact of the expression formalism. They also allow more flexible editing than finite-state machines. When one introduces a new state into a finite-state machine, one must specify entry and exit transitions, or the state will remain inaccessible and useless. One can simply enter new messages in a message-flow diagram by drawing them in their proper place.

6. Enhancements to message-flow diagrams

The enhancements to message-flow diagrams for the purpose of creating an unambiguous visual language fall into two categories. The first is that every event point contains an indication of *why* the event occurred; graphically if possible, textually otherwise. The second category of enhancement concerns the structure of the PEs. Some of this structure information is deducible from the diagrams themselves (for example, a message sent between two entities implies a connecting link), but even this information is better indicated explicitly, where it is apparent to the user, and where it can be used to enforce safety considerations, as when a message is sent between two entities when no connecting link between them has been declared.

6.1. History

One of the features that allows the programmer to indicate why an event occurs is the *history* facility. In figure 5a, the message-flow diagram fragment shows three event points, each indexed by a number.

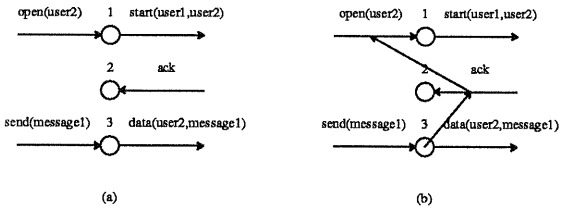


Figure 5 - history references

It is not clear from the diagram whether event 3 depends solely on the receipt of the **send** message (message receipts to an event are *always* part of the enabling condition), or whether it also depends on the acknowledgement or the **open** message having been received previously, or on the **start** message being sent, or on some combination of the three. Also, if it depends on some combination of those messages, should those messages have been received in some sequence? Such questions can be resolved by adding annotations referring to previous message history. Figure 5b indicates that event 3 depends on the previous receipt of the **ack** message, and, before that, on the receipt of the **open** message.

There remains the question of exactly what the history relationship is. For example, if event 3 depends on any previous **ack** message, that is different from the event depending on an **ack** that must occur in the immediately previous event. Such history references must be annotated: if the **ack** must occur in the previous event, the annotation is -1. If it must occur sometime before the previous event, the annotation is < -1. If it must occur no more than two events ago, the annotation is ≥ 2. (There may be both an upper- and lower-bound annotation.) Absence of any annotation indicates that the event may occur any time in the past.

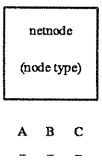
Each reference in a chain of history references may be annotated; the annotations taken as a whole, plus the sequence indicated by the chain, forms a set of constraints. It is the responsibility of the user to insure that the constraints are consistent, or the event will never take place.

Through testing, we hope to discover whether this annotation system is powerful enough to cover all cases that may appear. If not, we shall attempt to develop a richer system.

6.2. State variables and facts

There are situations where it is desirable to record information internally, as state information. Counting, for example, is best done using variables. We provide two facilities for this: state variables and state facts.

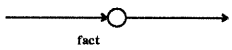
State variables are untyped variables associated with a given type of PE. Each PE of that type gets a full set of that type's state variables. They are declared as part of the PE type declaration. (See the discussion of PE type declarations, below.) They are shown at the head of the message-flow diagram below the PE's box:



State variables are referenced in *guards* and *associations*, and assigned through *actions*.

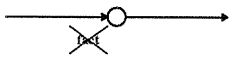
State facts are pieces of information that are *asserted* to be true at various places in the protocol. They may also be *retracted* (i.e., made no longer true). Facts may be any piece of text, although they generally have a structure form, similar to that of Prolog structures[10].

They are asserted by being placed after the event point that asserts them:



fact is true after the event.

They are retracted in a similar way:



fact is no longer true after the event (regardless of whether or not it was true before).

Facts are tested as parts of *guards*.

6.3. Timing

Timing considerations are important in the specification of protocols. Timeouts are used to break deadlocks, or to resend a message than has not been acknowledged. Timing information may also be used as a constraint, specifying that an event may not occur too long after another event.

Timing information in our language resembles history information. If an event must occur after a certain amount of time (generally, a number of seconds), or within a period of time, or any time after a period of time, a *timing connection* is placed between the two events in question, and is annotated with *t* **sec**, $\leq t$ **sec**, or $\geq t$ **sec**, respectively. Timing may also refer to event counts rather than actual time. In this case, the timing connection has an annotation of the form *t* **events**.

Figure 6 specifies that event two must occur two seconds after event 1.

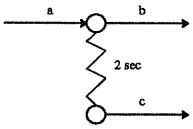


Figure 6 - timing connection

6.3.1. Textual guards and actions

There are a number of situations where it is more appropriate to express the reason that an event takes place in terms of text than visually. Conditions involving state variables and facts fall into this category. Such conditions are known as *textual guards*, and they function in conjunction with such visual enabling conditions as message receipt, history, and timing. Textual guards are boolean expressions involving state and local variables, and appear in *windows* associated with the event points. These windows generally are hidden, but are exposed when the event point is selected with a mouse. When the window is hidden, the associated event point is highlighted to indicate that there are associated textual conditions.

Figure 7 shows an event with a window indicating that the rule should only fire when, in addition to any other conditions, the state variable A is between 3 and 7.

There is also a *textual action* associated with this window. Such actions allow the event to modify the values in state variables. In the example given in figure 7, the action indicates that A should be incremented.

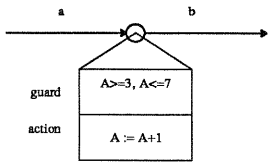


Figure 7 - textual guard and action

6.4. Variable associations

A specification is often more understandable when a specific example of a message is shown instead of the general message template, or when a variable name masquerades as a specific value (as occurs in Query-by-Example[24]). For example, in figure 8a, the message **data**(3,Message) is sent. Capitalized variables, such as message, are variables by convention and are assumed to refer to the same value wherever they appear. One may consider the word *Message* to be a prototype value rather than a variable name, however. The value 3, on the other hand, is problematic. It is not clear whether the value in that position is always 3, or if 3 is a derived value, and in this case is only a specific instance of a more general function. It might represent a message sequence number, for example. In such cases, a window is associated with the variable in question, containing the function that derives the value. Such association windows may also indicate how a value is used; that is, with which variable the value is associated in the receiving PE. Like windows associated with events, windows associated with values may also be hidden, and when they are, the associated value is highlighted. A value may have multiple associations, each indicating a different use or derivation. Figure 8b shows the window associated with the value 3 in the message. It indicates that the value 3 is really the value of the state variable Seq, which must belong to the PE from which the message originated.

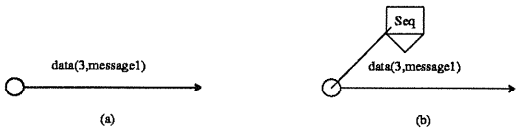


Figure 8 - value windows

6.5. PE and configuration declarations

The second class of enhancements allows us to declare the structure of types, or classes, of PEs. In particular, we must name the type and indicate the ports and variables inherent in that type. Ports may be input or output, and may have multiple instantiations. Figure 9 shows a PE type declaration. inport and outport are input and output ports, respectively. portArray is an array (with an unspecified number of elements) of output ports. The PE type also has three state variables, A, B, and C, with initial values.

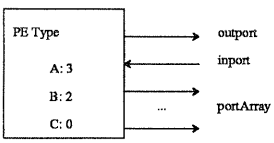


Figure 9 - PE declarations

Every protocol is specified on a sample configuration. Such configurations must be declared and indicate whether or not a message may be exchanged between two PEs. PEs may be connected by multiple links, in which case any message arrow between the two PEs in question must be labeled with the name of the link on which it travels.

Figure 10a shows a configuration in which two PEs have multiple connections. Figure 10b indicates on which link the message actually travels.

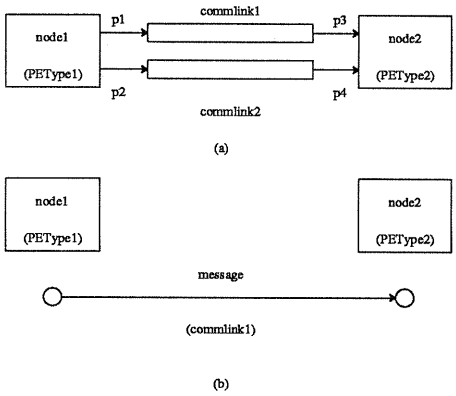


Figure 10 - configuration declaration and labeled message

7. Example: specification of a sliding window protocol

Figures 11 and 12 show a graphical specification of a sliding window protocol. The protocol is described in[13] and the diagram is adapted from an ambiguous Cara message-flow diagram described in[9]. Each diagram describes a scenario, the first being that which occurs when no messages are lost, and the second covering the case where messages are lost. We will describe the two diagrams line by line. (The line numbers are given on the left margin of the diagrams.)

Before we describe the diagrams, however, we will first briefly explain the way in which the diagrams are interpreted. Each event point along, with its associated guards, actions, history and timing connections, and value associations, is translated into a rule of behavior in a database which is associated with the PE's type. If a group of event points all denote the same rule, that rule is deposited only once, and all subsequent events denoting that rule have no effect; they are, in effect, comments. Thus, from the point of view of the language interpreter, the actual order of the various event points does not matter, apart from the fact that events explicitly connected by history or timing must be presented in the proper order. Aside from this, the order of event points and message exchanges in the diagramsis chosen by the designer from the standpoint of aesthetics, clarity, and readability. Thus, although diagrams may be validly read as typical legal scenarios under the protocol specification, they are actually translated in a somewhat different way.

In both diagrams, we have two nodes of type user, and two nodes of type netnode. We are concerned here with the specification of the behavior of the netnodes, since they enforce the protocol. Netnodes have state variables S, R, A, and E, although netnode1 is playing the role of the sender and we need only display variables S, R, and A, and netnode2 in the role of receiver need only display variable E. Initial values of S, R, A, and E are set to 0, 0, 0, and 1, respectively.

In the first event of figure 11 (line 1), netnode1 receives a **send** message. The message has an argument denoted by the symbol D1. Because the number of outstanding messages (S-A, or the sequence number minus the number of the most recent acknowledged message) is smaller than the window size (in this case, WINDOW=3), netnode1 transmits a **data** packet including the received value D1 and the value 1, which happens to be the sequence number. In addition, netnode1 increments the sequence number S, and adjusts the number of the message we may have to resend, R, to be the maximum of the current R value, and the number of the message following that of the last message acknowledged. On the same line, netnode2 receives the message, but only if its sequence number is equal to the number of the expected message E. In that case, we relay the message and increment the E variable. In all cases, we show the current values of the state variables. This is optional and may be suppressed.

On the second line, netnode2 spontaneously (the **true** condition means that it may happen at any time) sends an acknowledgement. The message number it acknowledges is one less than the current expected message number (i.e., the previous message received). That value is also bound to netnode1's variable A. (A and E are associated with netnode1 and netnode2, respectively, because they are the only nodes in which such variables are declared. If there is an ambiguity, we can differentiate it by qualifying the variable names.)

Lines 3, 4, and 5 add no new information to the protocol; they simply serve a documentary purpose. Each netnode1 and netnode2 action makes use of the previously defined behavior, which is used here simply to document the behavior of the protocol. Likewise the actions on line 6 use already existing rules to update the state variables. In line 6, since netnode2 expects message 5, we are free to acknowledge message 4. This updates variables in netnode1 indicating that we have received acknowledgements for all messages up to 4, and that we could retransmit message 5. In lines 3 through 6, the windows defining these actions are suppressed, although the user has the option of displaying them.

Figure 12 indicates the protocols actions if messages are lost. In lines 1 and 2, netnode1 transmits messages according to previously established rules, but the messages vanish into ''black holes'' In line 3, netnode1 spontaneously retransmits message 1 if there are outstanding messages (i.e., S-A > 0). This could happen any time the conditions hold, although it would be possible to add timer information. The message is received as previously specified.

The diagram then shows what happens if a message is received by netnode2 other than the one expected. In line 4, the previous message is acknowledged, but the acknowledgement does not arrive until netnode1 decides in line 7 to retransmit the as yet unacknowledged message 1. The message is received by netnode2, but because it doesn't match the expected message, it is ignored. Finally, the acknowledgements arrive and the variables are updated.

The defined event points describe most occurrences in the protocol: namely, what happens when

1) a message is transmitted,

2) a message is retransmitted,

3) a message is acknowledged,

4) the acknowledgement is received,

5) a message other than the expected one is received,

6) an expected message is received.

One situation not handled is that when netnode1 gets a **send** message, but there are no slots left in the window. It is possible that the message is ignored, or that it is buffered. Either solution only requires a few additional lines on the message-flow diagram.

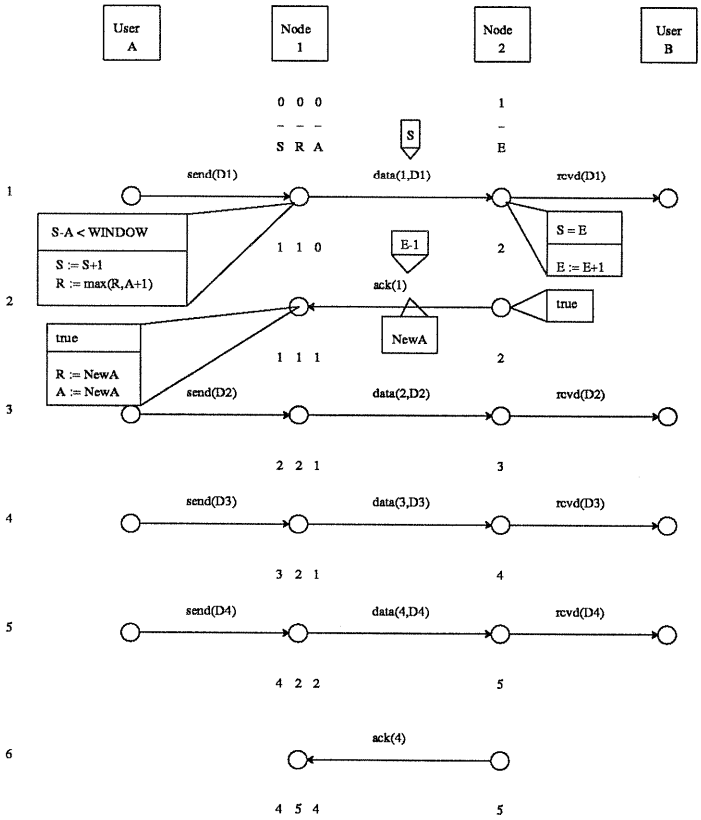


Figure 11 - sliding window protocol (I)

8. Additional issues

8.1. Multiple views - hiding guards and actions

In a complex diagram, a large number of windows containing textual guards and actions can clutter the diagram and make it unreadable. In order to prevent this, it is possible to hide the windows and suppress the display of current values of state variables. Any event point or value that has a window hidden behind it is highlighted. Pointing and clicking with the mouse will cause the window to be displayed.

When the diagrams are displayed on paper for use as documentation, neither the displayed window format nor hidden windows are appropriate. Instead, the diagrams are displayed using a *footnote format*. Any event or value with an associated window is annotated with a number, and a footnote corresponding to the number and containing the window text is displayed at the bottom of the diagram.

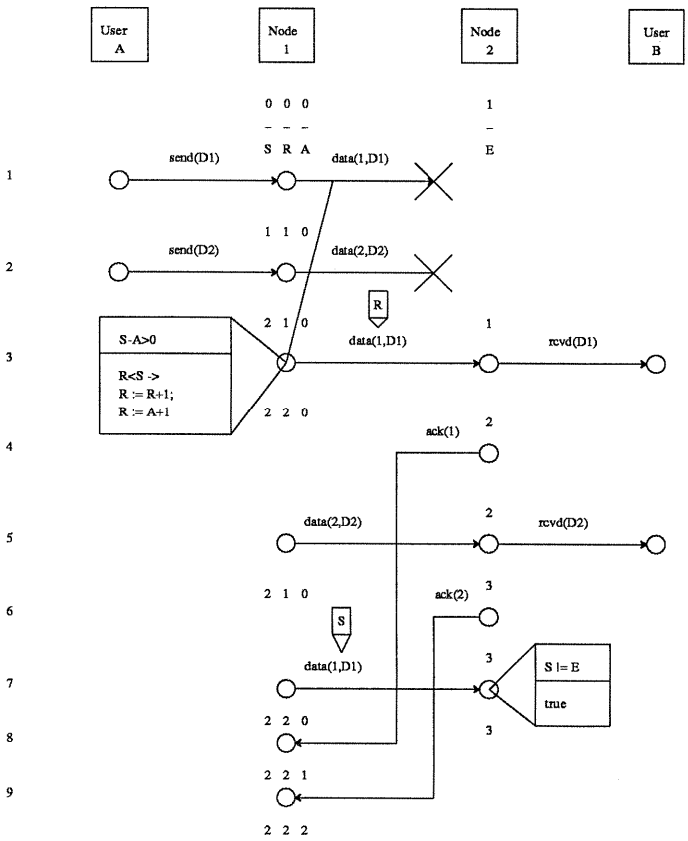


Figure 12 - sliding window protocol (II)

There is also *output-behavior format*, that only displays message exchange and suppresses all other information.

8.2. Debugging through corrected visualizations

As mentioned in a previous section, one of the strengths of having the language use the same model as the visualization is that a very powerful form of debugging may be used.

Using the message-flow diagram language, debugging follows the following procedure. First, previously declared PE types are used to create a new test configuration. Then, initial conditions are specified through the drawing of message arrows and the entering of initial values for variables. The protocol is then simulated. If the protocol is incomplete or ambiguous, the simulation will suspend and the system will query the user for the next step. The final result is a message-flow diagram with the same features as the initial program. Event points with associated textual conditions and actions, and messages with values derived from functions, will have associated windows (which will be hidden until they are called up). The simulation output, therefore, can also be considered a graphical specification, albeit one which adds no new information to the specification of the protocol. If it shows an incorrect or unanticipated behavior, it can be edited to display the correct behavior and recompiled, the corrected events superseding the old versions. This technique of debugging by manipulating program output promises to greatly increase programmer productivity and will be the object of further research.

8.3. Modularity and multi-level specifications

Modern communications protocols are specified using a multi-level approach. This may be reflected in message-flow diagrams in two ways. The first is to specify each level separately. Figures 13a and 13b show a specification first at a high level, then show the underlying message exchange needed to convey a single high-level message.

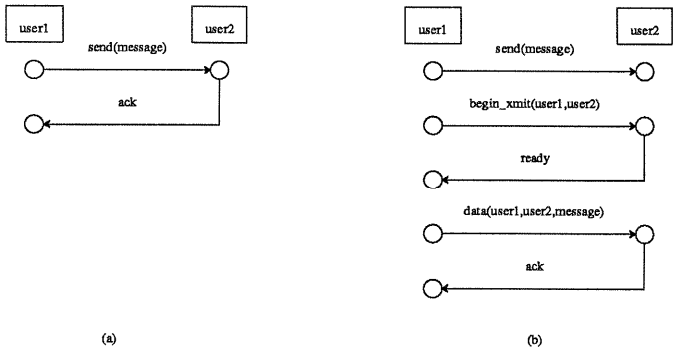


Figure 13 - multi-level specifications (I)

Each message exchanged in figure 13b may also correspond to a lower-level exchange.

The other approach is to flatten out the multiple levels on a single diagram, so that several specification levels of each PE are shown. Figure 14 shows such a diagram for the previous exchange.

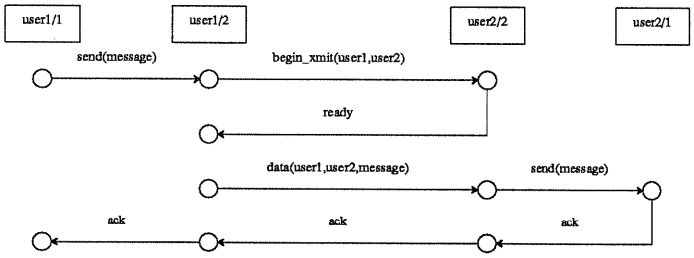


Figure 14 - multi-level specifications (II)

9. Future work

The use of message-flow diagrams for communications protocol specifications raises a number of research issues that are relevant to the larger issue of visual languages. The first issue concerns the specification of negative conditions. When an event depends on some previous message or event, connecting the events with some visual dependency relation (such as the history reference) is a natural visual model. However, problems arise when we wish to say that an event depends on something *not* happening. Since the initial event is not in the diagram, one cannot draw a reference to it. On the other hand, to insert the non-event into the diagram, even if it is shaded or otherwise distinguished as a non-event, is misleading, since it is a natural tendency for the reader to assume the existence of objects on the diagram, even if their representation indicates otherwise. Currently, our solution is to specify such conditions textually. Every graphical construct in our language has a textual equivalent, which may be used in a textual guard or action. In this case, we preface the condition with a **not** operator.

Although the syntax of our language is currently described in English, and the semantics are described in terms of the underlying Carla rule language[9], we plan to formally describe the syntax of our language. Visual language syntax is an area in which relatively little work has been done. Certain approaches, such as picture layout grammars[16], seem promising.

We are just beginning implementation of the message-flow diagram language. While we will probably use an augmented version of the Cara system[11] for our first implementation, we hope that work in this area will motivate basic research on the structure and specification of compilers for visual languages.

10. Conclusions

The visual language we have described indicates the usefulness of a visualization-based approach in language design. This usefulness stems both from the ability to solve problems in terms of suitable underlying mental models, and also from the ability to debug programs by recompiling corrected visualizations. Both these factors should improve programmer productivity and reduce errors.

Our experience designing this language has given us some insights into the interaction of text and graphics in a visual language. Adding textual guards and actions allows us to easily incorporate information into specifications that is not naturally graphical. A completely graphical language, or at least a language whose semantics are graphics-based, remains a goal, however. Graphical transformation systems such as ChemTrains[4] offer one possibility, and future versions of our language may rely on such completely visual approaches.

Acknowledgements

Many of the ideas for this work derive from work that the author did as a post-doctoral researcher at the IBM Zurich Research Laboratory in Rueschlikon, Switzerland, as a member of the Cara group. The author would like to thank the other members of the Cara group, and particularly Alistair Cockburn, for their comments, ideas, and conversations. The author would also like to thank Willibald Doerringer of IBM Zurich for several long discussions on the nature of visual programming.

The author would also like to thank Clayton Lewis and his research group (Brigham Bell, John Rieman, Bob Weaver, and Nick Wilde) for introducing me to the notion of doctrine, and for subjecting the language design to one of their grueling walkthroughs. Several of their suggestions have been incorporated into this paper.

References

1. *Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2,* Document SC30-3269-3, IBM Corporation, December 1985.

2. *The Formal Description Technique Estelle,* North-Holland, Amsterdam, 1989.

3. A. L. Ambler, "Forms: Expanding the Visualness of Sheet Languages," *Proc. 1987 Workshop on Visual Languages*, pp. 105-117, Linkoping, Sweden, August 1987.

4. B. Bell, J. Rieman, and C. Lewis, "Usability of a Graphical Programming System: Things We Missed in a Programming Walkthrough," *CHI '91 Conference Proceedings*, New Orleans, April 27

- May 2, 1991.

5. T. Bolognesi and D. Latella, ''Techniques for the Formal Definition of the G-LOTOS Syntax,'' *Proc. 1989 IEEE Computer Society Workshop on Visual Languages*, pp. 43-49, Rome, October 1989.

6. T. Bolognesi and E. Brinksma, ''Introduction to the ISO Specification Language LOTOS,'' *Computer Networks and ISDN Systems*, vol. 14, pp. 25-59, 1987.

7. T. Y. Cheung, Y. C. Ye, X. Ye, and G. Q. Wang, *UO-GLOTOS: A Model/System for Representing, Editing and Translating Graphical LOTOS,* Dept. of Computer Science, University of Ottawa. technical report

8. W. Citrin, *Inference-Based Simulation of Communications Architectures.* Submitted to 1992 ACM Symposium on Applied Computing, Kansas City

9. W. Citrin and A. Cockburn, ''An Executable Specification Language for History-Sensitive Systems,'' *IBM Zurich Research Laboratory Research Report*, no. RZ 2162, July 1991.

10. W. Clocksin and C. Mellish, *Programming in Prolog,* Springer-Verlag, Berlin, 1981.

11. 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.

12. M. Diaz and C. Vissers, ''SEDOS: Designing Open Distributed Systems,'' *IEEE Software*, pp. 24-32, November 1989.

13. R. Duke, I. Hayes, P. King, and G. Rose, ''Protocol specification and verification using Z,'' *Protocol Specification, Testing, and Verification, VIII*, pp. 33-46, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1988.

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

15. E. P. Glinert and S. L. Tanimoto, ''Pict: An Interactive Graphical Programming Environment,'' *Computer*, pp. 7-25, November 1984.

16. E. J. Golin and S. P. Reiss, ''The Specification of Visual Language Syntax,'' *Proc. 1989 IEEE Computer Society Workshop on Visual Languages*, pp. 105-110, Rome, October 1989.

17. J. Grabowski and E. Rudolph, ''Putting Extended Sequence Charts to Practice,'' *SDL '89: The Language at Work*, pp. 3-10, Elsevier Science Publishers B.V. (North-Holland), 1989.

18. K. M. Kahn and V. A. Saraswat, ''Complete Visualizations of Concurrent Programs and Their Executions,'' *Proc. 1990 IEEE Computer Society Workshop on Visual Languages*, pp. 7-15, Skokie, IL, October 1990.

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

20. D. New and P. Amer, ''Adding Graphics and Animation to Estelle,'' *Information and Software Technology*, vol. 32, no. 2, pp. 149-161, March 1990.

21. L. Snyder, ''Parallel Programming and the Poker Programming Environment,'' *Computer*, pp. 27-36, July 1984.

22. A. S. Tanenbaum, *Computer Networks,* Prentice-Hall, Englewood Cliffs, 1981.

23. K. Tsuda, A. Yoshitaka, M. Hirakawa, M. Tanaka, and T. Ichikawa, ''IconicBrowser: An Iconic Retrieval System for Object-Oriented Databases,'' *Journal of Visual Languages and Computing*, vol. 1, pp. 59-76, 1990.

24. M. M. Zloof, ''Query-by-Example: A Data Base Language,'' *IBM Systems Journal*, vol. 16, no. 4, 1977.