Design Considerations for a Visual Language for Communications Protocol Specifications

Wayne Citrin
Department of Electrical and Computer Engineering
University of Colorado, Boulder, USA

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 problem of specifying communications protocols has long occupied that branch of computer science concerned with computer communications and networks. 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. 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[1], based on extended finite-state machines, and LOTOS[3], 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.

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, 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[2], 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.

2 Previous efforts

GROPE (Graphical Representations Of Protocols in Estelle)[12], 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.

LOTOS, as mentioned earlier, is based on process algebra, and has no generally accepted visualization. Two visual ver-
Figure 1 - a GROPE visualization (from [12])

Figures 2a and 2b provide some examples of typical G-LOTOS and UO-GLOTOS code, respectively.

Another proposed visual representation is the extended sequence charts (ESC's) associated with the SDL language[10]. 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[8] 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.
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.

3 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.

The language should be static. In a number of visual languages, particularly those that are demonstration-based, 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[11].) 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[8].

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.

4 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 and textbooks. 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 PE’s column, or along message transmission arrows. The former condition ensures that an event point occurs before any of the
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.

![Message-flow diagram](image)

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.

5 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.

5.1 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.

![History diagram](image)

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 $\geq 2$. (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.

5.2 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 $> t$ 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](image)

5.3 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 7 shows a PE type declaration. Import and output 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.

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 7 - PE declarations](image)

5.4 Other enhancements

Lack of space prevents us from describing in detail the remaining enhancements, but they all concern the integration of text with the message-flow diagrams. These features include state variables and state facts, which allow information to be recorded locally within a PE to be referenced in later rule firings. They also include textual guards and actions, which allow the expression of certain conditions or actions which may be difficult or inappropriate to express graphically (for example, certain mathematical operations). Finally, they include methods for associating the contents of messages with the textual rules belonging to a PE, and for indicating how certain contents of messages are derived. Integration of textual specification into the diagrams is a significant feature of our work.

Some of these features are described in the example in the next section. The remainder, along with the rest of the language, is described in more detail in[6].

6 Example: specification of a sliding window protocol

Figures 9 and 10 show a graphical specification of a sliding window protocol. The protocol is described in[9] and the diagram is adapted from an ambiguous Cara message-flow diagram described in[7]. 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. 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 diagrams is 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 9 (line 1), netnode1 sends a 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 10 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.

Figure 9 - sliding window protocol (I)

7 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 \texttt{not} operator.

A second issue we are investigating concerns the use of corrected visualizations in the debugging of the program. As mentioned earlier, we expect that such a technique would simplify debugging dramatically.

Finally, our language currently only specifies flat protocols. We are currently investigating methods of specifying multi-level protocols.

We are just beginning implementation of the message-flow diagram language. While we will probably use an augmented version of the Cara system\cite{8} 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. We also plan to formalize our language specification.

**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.

I 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**