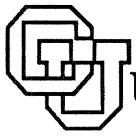
Simulation of Communications Protocols through Graphical Transformation Rules

Brigham Bell, Wayne Citrin

CU-CS-597-92 June 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Simulation of Communications Protocols through Graphical Transformation Rules

Brigham Bell

Wayne Citrin

CU-CS-597-92

June 1992

Brigham Bell
USWest Advanced Technologies
Suite 270
4001 Discovery Drive
Boulder, CO 80303

bbell@advtech.uswest.com

tel: 1-303-541-4000

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

Simulation of Communications Protocols through Graphical Transformation Rules

Brigham Bell

U S West Advanced Technologies, Suite 270
4001 Discovery Drive, Boulder, CO 80303, USA

Wayne Citrin

Department of Electrical and Computer Engineering, Campus Box 425
University of Colorado, Boulder, CO 80309-0425, USA

ABSTRACT

We present a novel approach to the specification and simulation of communications protocols, through graphical transformation rules. These rules are expressed using ChemTrains, a graphical transformation system. For the protocols tested, the sets of rules are concise and easily constructed. Such a system should be suitable for rapid prototyping, performance analysis, and instruction. Whether executable protocol code can also be generated from the graphical transformation rules remains an open question.

1. Introduction

Over the past five years, a number of efforts have been made to construct visual representations of communicating systems. Some of these systems aim to animate or trace the message-passing behavior of such systems ([17], for example); others have been designed to allow the programmer to specify the communications rules, or protocols, graphically[5,7]. Still others have been designed to help the user manage the complexity of protocols as large software structures (PegaSys[16] has been used for this, although it is not specifically intended for this purpose), or to assist the user in setting up and managing network configurations[11].

There are a number of reasons for these efforts. Communications protocols are specifications of complex non-sequential systems, for which textual representations, being inherently sequential, are difficult to read, write, and understand. Visual languages may provide a more natural match to the system being described, reducing the cognitive effort of both reader and writer. (Textual specification methods for such systems are also an active area of research[9].) A further reason for the interest in visual specifications in this field is that the communications domain already possesses a strong set of visualizations, including state transition diagrams[17], message-flow diagrams[8], protocol towers[19], and network diagrams[11]. From the point of view of visual language researchers, communications protocol design provides the opportunity to apply research ideas to a large, complex domain in which "real-world" problems will be solved, and in which experts in the domain will use the tools.

One common thread among the visual specification schemes that have been developed or proposed is that they are all based on some underlying textual form. For example, the state transition diagrams of the GROPE system[17] (which, strictly speaking, is a system for visualizing protocols rather than for specifying them, but from which a visual specification method could quite naturally be derived) correspond to the underlying Estelle[10] specification. The motion of the token representing the locus of control from state to state is based solely on the actions dictated by the underlying textual specification, although the visual analogues are suggestive.

The G-LOTOS language[5] is a visual version of the textual LOTOS system, designed to simplify the syntax. No semantics is suggested by the graphics at all. The MFD system[7] which uses message-flow diagrams, does attempt to provide suggestive graphical representations for they underlying semantics, but although the graphical form was developed before the underlying Carla textual representation[6], all the semantics are ultimately based on the underlying text.

The point is that the graphical behavior of all of the above specification methods derives from the semantics of the underlying text and not from any inherent properties of the graphics themselves. The graphics may serve to elucidate the meaning of the text, but the user of the system generally has to be familiar with the underlying textual representation in order to make full use of the method.

The result is that the user of such graphical systems must now be aware of not one but two (albeit related) specification systems. This added burden is at odds with the original intent of easing the user's cognitive burden by employing visual representations.

In order to recover these original advantages, we have decided to employ a completely visual approach, in which the specification semantics are entirely contained within the graphics. We do not go as far as Kahn's "complete visual programming" [14], in which every picture is both a snapshot of the program state *and* a complete representation of the program itself. In our approach, the program state and the graphical transformation rules are maintained separately.

The ChemTrains system [2,3] (described below) is a general-purpose graphical rule-based language in which graphical entities on a screen determine the state of an executing program. The program consists of graphical transformation rules (pattern/result pairs) that the system attempts to match to some portion of the state. Once a match is found between the pattern of a rule and part of the state, the matched entities are transformed to conform to the result of the rule. This transformation of the state constitutes a program step.

In using ChemTrains to simulate network communications, the graphical program state represents the state of the network. This includes the entities and connections in the network itself, as well as the state of those entities and connections (message queues, internal variables, messages en route, and so forth). The graphical transformation rules represent rules of behavior that the protocol must obey. When the system, as displayed graphically, is in a certain state, and the pattern part of a transformation rule matches some part of that state, the state is transformed in accordance with the result part, thus reflecting the execution of a step of the protocol. In addition, some transformation rules describe the properties of the communications media; chiefly, their speed, noisiness, and reliability. Random behavior can be simulated through probabilistic application of rules.

In order to show how graphical transformation rules may be used to specify communications protocols, we provide examples of two relatively simple protocols, the alternating bit protocol and the sliding window protocol, and describe the way in which they may be expressed through ChemTrains transformation rules. Although these protocols are simple ones, we believe that more complex protocols may be expressed through the same or similar techniques. The technique we describe is suitable for rapid prototyping and performance evaluation, as well as for the illustration of protocol operation for instructional purposes.

2. ChemTrains language description

ChemTrains is a general-purpose visual programming language for describing graphical simulations that have a qualitative behavior model, such as document flow in an organization, the phase change of a substance as temperature varies, or a Turing Machine. ChemTrains models show objects participating in reactions similar to chemical reactions and moving among places on the screen along paths. The name "ChemTrains" was suggested by the chemical reactions and the role of paths, thought of as train tracks. The

system was built in Macintosh Allegro Common Lisp[1] and works as a standalone application on Macintosh computers running version 6 of their operating system.

The ChemTrains programming environment enables a graphical programmer to draw a simulation as it would initially appear to an end user, and then to specify the behavior of that simulation by drawing graphical rules. Each rule has two main components: a pattern picture and a result picture. When a simulation is executed, ChemTrains uses a rule interpreter to animate the display. When the rule interpreter recognizes that the pattern of one of the rules is identical to a portion of the simulation display, it then replaces that portion of the display with the picture in the result of the rule. When trying to recognize whether a pattern matches a portion of the display, the interpreter decides based on whether the objects in the pattern match corresponding objects in the display, and the topologies of the pattern and the display also match. The display will continually simulate as long as a pattern of a rule matches part of the display. When none of the rule patterns match the simulation, the rule interpreter stops.

The model of computation is a production system or rule-based model, introduced by Newell as a way to describe simulations[18]. Since then, production system languages such as OPS5[12] have been developed and used in building expert systems. More recently, Furnas has demonstrated with the BITPICT system the use of graphical rewrite rules to build graphical simulations[13]. A BITPICT rule modifies a graphical display when the pattern picture of the rule exactly matches the pixels of a portion of the display. The difference between BITPICT and ChemTrains is that while a BITPICT rule matches a portion of the display when the *pixels* and their *geometry* match, a ChemTrains rule matches a portion of the display when the *objects* and their *topology* match. The following language description defines terminology with a simple house lighting simulation and then describes the semantics of the rule interpreter also with this simulation.

2.1 ChemTrains terminology

A *simulation display* is a window that displays a graphical simulation. Figure 1 is an example simulation display. ChemTrains rules operate on the simulation display similar to the way that rules in a production system language operate on the working memory.

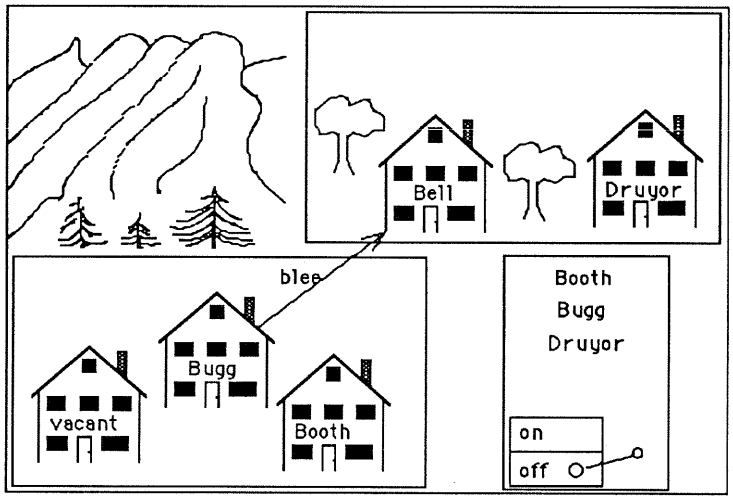


Figure 1 - Example simulation display

An *object* is a graphical object of the display. An object may either be a rectangle, an oval, a polyline, a text string, or an icon. Figure 1 shows a picture in which each kind of object is used. The mountain scene and each of the houses are a single icons. Five rectangles are shown, two that are used to group houses, and three that are used to compose a drawing of a switch at the bottom right side of the picture.

Two objects are *identical* if the objects are the same kind (either rectangle, oval, polyline, icon, or text string) and they have the same bit pattern in the display. All of the houses in figure 1 are identical because they are all copies of the same house icon. The two large rectangles are also identical, because they are both rectangle objects with the same dimensions. Two objects that look identical may not actually be identical. For example, two rectangles may not be identical if one was constructed as a rectangle object and the other was constructed out of line segments.

An object *contains* another object if its bounding rectangle encloses the bounding rectangle of the other object. In figure 1, the houses each contain text string labels. A *container object* is an object that is being used in the simulation to contain other objects. Any object may be a container object. The houses are container objects for their labels. The two small boxes containing the "on" and "off" text strings are also container objects.

A *path* is a line connecting one object to another object and can either be *directed*, displaying an arrowhead on one end, or *non-directed*, having no arrowhead.

A *replacement rule* or *rule* is a mechanism for defining general changes in a picture. A replacement rule has a name, a pattern picture, and a result picture. When all the objects of the pattern picture of a rule match objects in the simulation picture, the matched objects are replaced with objects shown in the result picture of the rule. Figure 2 shows an example rule called "house on."

A *variable* is an object in either the pattern or result of a rule that may represent any object in the simulation. When an object is variablized, it is marked with a big V or, if it is a text string, displayed in italics. In figure 2 the "name" text strings are variables, and everything else is a constant.

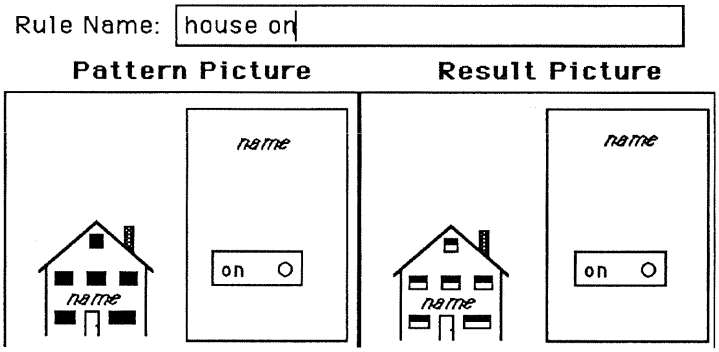


Figure 2 - Example transformation rule

The rule interpreter of ChemTrains, like other production systems, executes a recognize-act cycle that has three phases:

1. a *pattern matching* phase, in which those rules whose pattern matches the display are identified,
2. a *conflict resolution* phase, in which one of those rules is chosen, along with the objects in the display with which the pattern matches,
3. a *rule execution* phase, in which the actions of the selected rule are executed, altering the display.

The cycle then repeats until no match is found, in which case the system halts. Each of the three phases is elaborated below.

2.2 Pattern matching

The ChemTrains pattern matcher will match a pattern picture to a portion of the simulation display if and only if:

1. Every object in the pattern that is not a variable matches an identical object in the picture.
2. Every variable in the pattern matches an object in the picture.

3. Identical variables in the pattern match identical objects in the picture. For example, two identical variables in the pattern can match any two things in the simulation as long as the two things are identical.
4. Every containment relationship in the pattern must also be satisfied by the matched objects.
5. For every nondirected path in the pattern, a nondirected path must connect the matched objects.
6. For every directed path in the pattern, a directed path must connect the matched objects and point in the same direction.

Any other sort of geometric relationship between objects in the pattern picture, such as adjacency or distance, is ignored in pattern matching. For example, the pattern shown in figure 2 literally means: match an unlit house that contains some object, and match a rectangle identical to the large rectangle, containing an object identical to the object inside the unlit house and a rectangle identical to the small rectangle, containing a text string named "on," and an oval identical to the oval shown. When the switch in figure 1 is turned on, the pattern of this rule matches the picture in three different ways, matching the "Bugg," "Booth," and "Druyor" houses. The two other houses are not matched because the names inside the houses do not match names inside the rectangle.

2.3 Conflict resolution

When more than one rule matches a picture, the rule highest in priority is chosen to execute. The language environment supports a feature for ordering the rules. An example of using rule ordering to control a simulation would be in writing rules to play tic-tac-toe: a rule to play a win would be placed before a rule to play a block. When a rule is chosen and this rule may match multiple combinations of objects in the simulation, it chooses between the possible combinations randomly.

2.4 Rule execution

When a rule is chosen, the system executes the rule by interpreting actions, specified by differences between the pattern and result pictures of the rule. The following kinds of differences are permitted:

1. objects or paths from the pattern may be deleted;
2. paths may be added;
3. objects may be added if they are placed inside an object that is in both the pattern and result pictures; or
4. objects may be added if they replace a deleted object from the pattern picture, as the lit house replaces the unlit house in the "house on" rule shown in figure 2.

When the control knob is moved to the "on" position in the simulation display, the "house on" rule executes three times on consecutive recognize-act cycles (one for each unlit house), resulting in turning these lights on. Another rule can be added to this simulation to enable the house lights to be turned off when the switch is moved to the "off" position.

3. First example: the alternating bit protocol

3.1 Description of the protocol

The alternating bit protocol is a simple communications protocol designed to allow reliable data transmission over noisy telephone lines [15].

The protocol can be described informally and simply. Two communicating entities are involved. One plays the role of *sender*, and the other is the *reciever*. The sender and receiver are connected by two one-way communications links, one leading from sender to receiver, and the other from receiver to sender. (This is a half-duplex specification. The protocol can be generalized to full-duplex operation.)

The message packets sent by the sender contain the message itself, and a sequence bit. The acknowledgement packets sent by the receiver consist of a bit which is the sequence bit of the message being acknowledged. The sender has an internal bit that represents the sequence bit value to be used for the next message to be transmitted, and the receiver has an internal bit that represents the value of the sequence bit of the next message it expects.

The basic principle of operation is that the sender may not send the next message (assuming that there is one to send) until it has received an acknowledgement for the previous message (as determined by the bit value in the acknowledgement packet). It may, however, resend the previous message (i.e., the one it last sent) at any time. The receiver simply ignores any message that has a bit that differs from its current internal bit value, and sends acknowledgements at any time for the previous message it has received (that it didn't ignore). More rigorously, we can define the following rules of behavior for the sender:

1. If the first message is about to be sent, or if an acknowledgement has been received for the message with the previous bit value (i.e., the complement of the current bit value), then send a package containing the next message and the current bit value and flip the bit.
2. resend the previous message with the previous bit value at any time.
3. If an acknowledgement is received whose bit matches the current bit value, ignore it.

The receiver's behavior is defined as follows:

1. If a message packet is received whose bit matches the receiver's current bit value, then send an acknowledgement with the current bit value and flip the bit.
2. Resend an acknowledgement for the previous message at any time.
3. Ignore (and discard) any incoming message whose bit does not match the receiver's current bit.

Examination of the protocol specification will show that the protocol will work (i.e., succeed in sending all the messages in their proper order) as long as there is a non-zero chance that a message will travel uncorrupted over the link. The bandwidth may be very low, and there may be many redundant messages, but the goal of the protocol design is reliability, not efficiency.

3.2 Specification of the alternating bit protocol using ChemTrains

Specifying the alternating bit protocol in ChemTrains involves two parts: drawing the initial state (the network configuration), and drawing the rules of behavior.

Figure 3 is the ChemTrains diagram showing the initial configuration of the network. The three large boxes represent the communicating entities; they are labeled "fred," "wilma," and "ethel." Between the entities are smaller boxes representing the communications links. They are connected to the entities with arrows denoting direction of the message traffic and are labeled with the destination of messages traveling on that link. Within the communicating entities' boxes are sub-boxes containing additional information and structure. The sub-box in the upper-left corner of each entity's main box contains the name of the entity and the current value of the state bit (initially 1). The sub-box below that contains the entity's role (*send* or *receive*) and the entity to which messages or acknowledgements will be sent. Along the bottom of the sender's box is a buffer containing a chain of messages to be sent (in this case, a sequence of letters) and a pointer indicating the next message to be sent. The receiver contains a similar buffer for receiving messages, along with a pointer indicating where in the buffer the next message should be placed. Also in fred's box is a dummy acknowledgement message that serves to initialize the protocol. Recall that the specification discussed in the previous section includes a special case for sending the first message; by pretending that the sender has received an acknowledgement for a previous message, we can eliminate that case.

The diagram also contains a counter, which is incremented by a rule each time a message traverses a link. There is also a box containing the values 0 and 1. This is a

graphical idiom that allows us to compare and flip state bits; it will be explained later as it is needed in the rules.

Finally, there is a box near the top of the diagram which is divided into two parts, one of which contains an X. This is an idiom dictated by the pattern matching mechanism and is used by the rules governing message transmission.

It should be noted that the configuration in figure 3 permits communication between any two of the three entities, provided that the proper destination labels have been set. In this example, we only demonstrate the sending of messages from "fred" to "ethel."

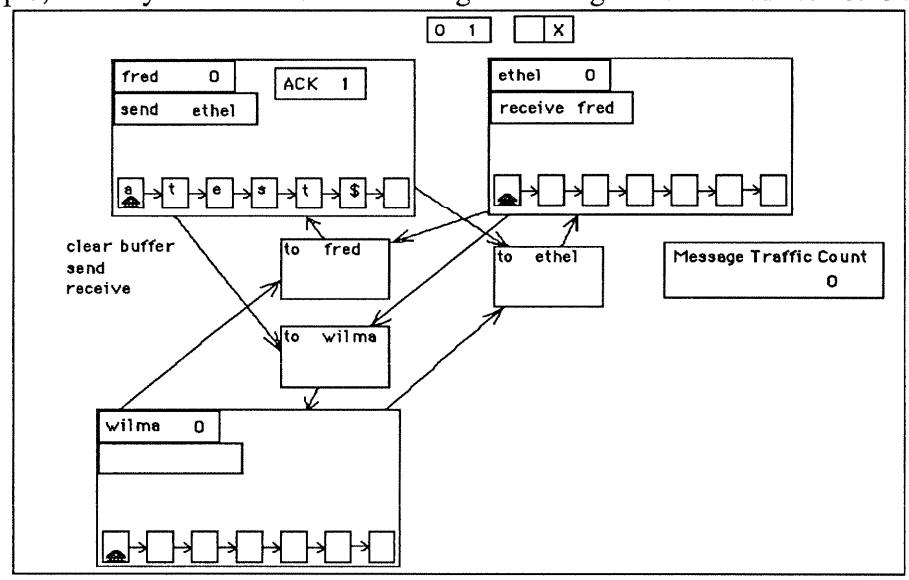


Figure 3 - Sample initial network configuration

Figure 4 shows the "send message" rule, which states that, if the buffer pointer occupies a space within a message (that is, if there is a message to be sent) and an acknowledgement has been received whose value is different from the current state bit value, then the acknowledgement message is removed, the buffer pointer advanced, the message and current bit value bundled and placed onto the link, and the state bit flipped.

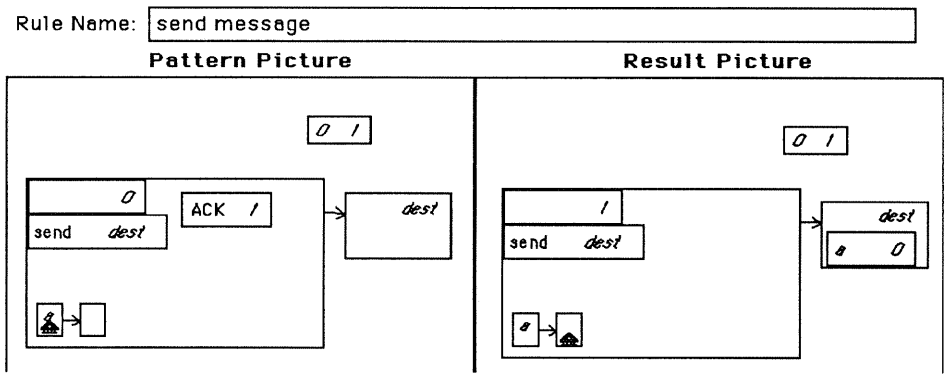


Figure 4 - Rule governing message transmission

Several interesting features of the rule should be noted. First, italicized tokens denote variables. Message *a* stands for any message that may be sharing a buffer box with the pointer token. All appearances of *a* in the rule denote that same token. Thus, *a* remains in the buffer as the pointer is advanced, and a copy of *a* is bundled with the bit value that was current at the time the pattern was matched. (This bit value is denoted by *0*, also a

variable.) Another use of variables is shown by the variable *dest*, which matches whatever shares a box with the "send" token. What is matched is the name of the message's destination, so the message packet is placed on a link labeled with that destination. (It is necessary to do this since there are two links departing each entity in our configuration.) Finally, the rule uses the box containing the 0 and 1 to compare the current bit and the acknowledgement value, and also to flip the bit. The pattern picture says that if the current bit value matches one of the values in the 0/1 box, the field of the ACK message must match the other value in the 0/1 box. (Thus, they must be different.) Then, the new value of the bit will be the same as the value of the acknowledgement packet, which is different from the previous state bit value, and the bit is flipped.

The rule "send acknowledgement" (shown in figure 5) records the contents of a new message into the buffer, advances the buffer pointer, sends an acknowledgement back to the sender, and flips the current bit, when a reciever has a new message, and the message's bit value matches the internal bit value (which is the expected bit value for the message).

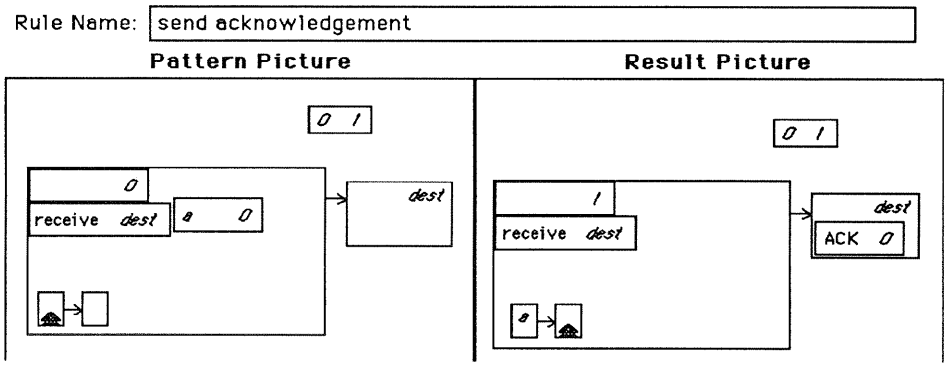


Figure 5 - Rule governing message reception and acknowledgement

The rule "ignore old message" (shown in figure 6) erases a message when it holds an incorrect or unexpected bit value. Another rule "ignore old acknowledgement" (not shown) similarly erases an acknowledgement that has an incorrect bit value.

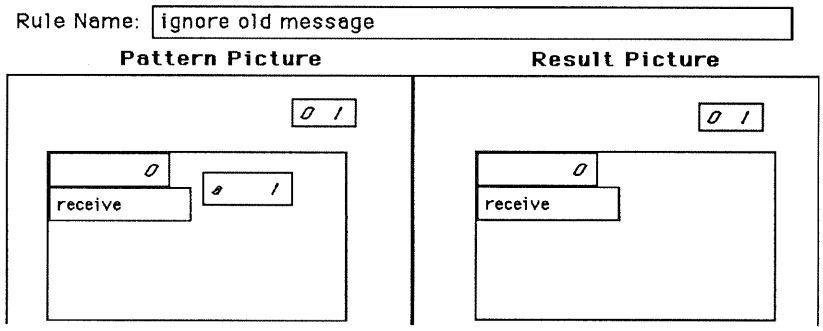


Figure 6 - Rule governing receipt of erroneous message

The rules "resend message" (shown in figure 7) and "resend acknowledgement" (not shown) are examples of the probabilistic specification of actions. We wish to allow the entities to resend the previous message or acknowledgement whenever a new message cannot be sent, thus breaking the deadlock. However, since the rules are ordinarily assigned priority by the order in which they appear in the rule base, this might indicate that one of these two messages would always be chosen to fire, and that the other one would starve. To avoid this, we assign priorities to these two rules so that they are considered to appear in the same place in the rule order, but that the rule "resend message" should be

fired with a probability of 0.5, and "resend acknowledgement" should also be fired with a probability of 0.5. There is then an even chance that either rule will fire.

The double box with the X is needed to allow a message to be resent more than once. The pattern matching mechanism of ChemTrains will not execute the same rule twice in succession on the same program state. However, this prevents the "resend message" and "resend acknowledgement" rules from firing twice in succession, which should be allowed. To force the configuration to change, and therefore allow a second firing, the retransmission rules switch the X between boxes when they fire. The configuration is now different, and the rule can fire again.

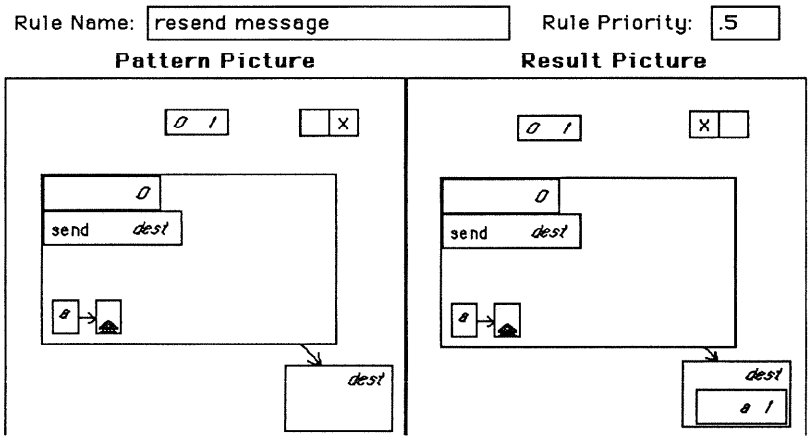


Figure 7 - Rule governing retransmission of message

Two rules named "sender done" and "reciever done" (not shown) are needed to terminate the protocol. When the buffer pointers have traversed over the end-of-message "$" token, these rules terminate the sending and recieving processes by deleting the destination address within the sending or recieving network entity.

These rules fully cover the semantics of the protocol but remain to define the behavior of the communications links. Although this is not part of the protocol proper, the rules defining this behavior are part of the rule base. In order to test our protocol's behavior on unreliable links, we define a link which loses the message 10% of the time. We do this by creating the rules for message transport, both of which will be applicable at the same time, and assign probabilities to them. The rule "continue message" (figure 8) will cause a message on a mail box to be relayed to the connected network entity with probability 0.9, while the rule "lose message" (figure 9) will cause the message to disappear with probability 0.1.

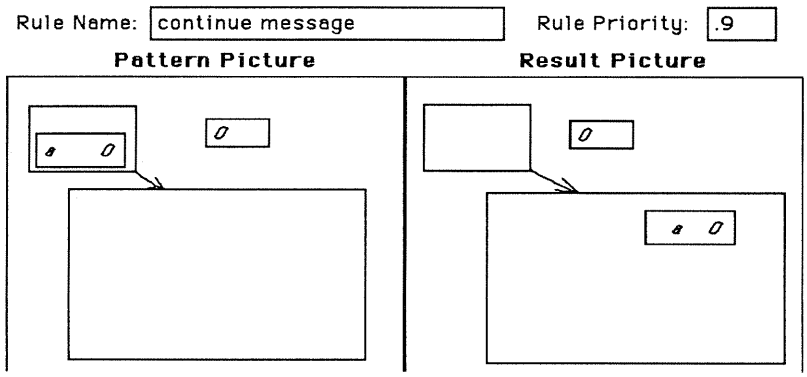


Figure 8 - Rule governing relaying of a message on a communications link

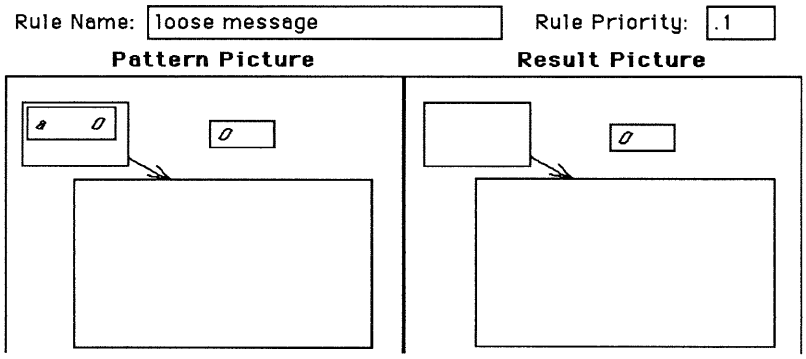


Figure 9 - Rule governing loss of message on a communications link

As mentioned before, the priority of the rules is defined by their order. Thus we must make decisions concerning the ordering of the rules. The main features of the ordering (shown in figure 10) are that rules concerning link behavior must have high priority so that the link will be clear for the next message, and that rules governing resending of messages must have low priority so that they are fired when no new acknowledgements may be sent.

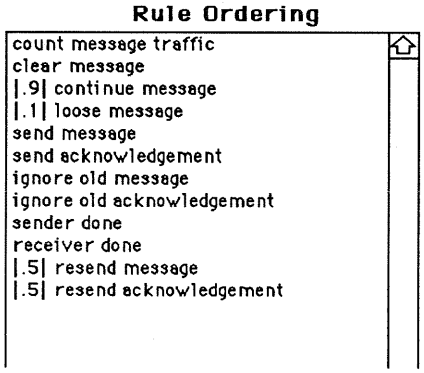


Figure 10 - Rule ordering of alternating bit protocol and network communication rules.

The two top rules in the ordering, "count message traffic" and "clear message" have not been shown here. The former rule simply increments the message counter whenever a message passes through a link. The latter rule simply allows the user to reinitialize the system by clearing the receiver's message buffer and resetting the buffer pointer back to the beginning of the buffer.

To run a simulation, one simply sets up a configuration like that shown in figure 3 and activates the rule firing. When the rule application speed is properly adjusted, one can see the messages traveling between the entities along the links, the message buffers filling and the pointers advancing, and the bits changing value. This is very useful both for instruction on the operation of the protocol and for rapid prototyping to test the validity of a specification. In addition, one may add counters to the configuration (there is one such counter in figure 3) and add rules that cause the counters to be incremented upon the occurrence of certain events. These may be used for performance evaluation. Figure 11 shows a snapshot of the network at an intermediate stage in the execution. The first two messages have been sent, and an acknowledgement for the second message is on its way back to the sender. Since the counter shows that five messages have been sent (including acknowledgements), some messages must have been lost on the links, but it appears that the protocol has handled this properly.

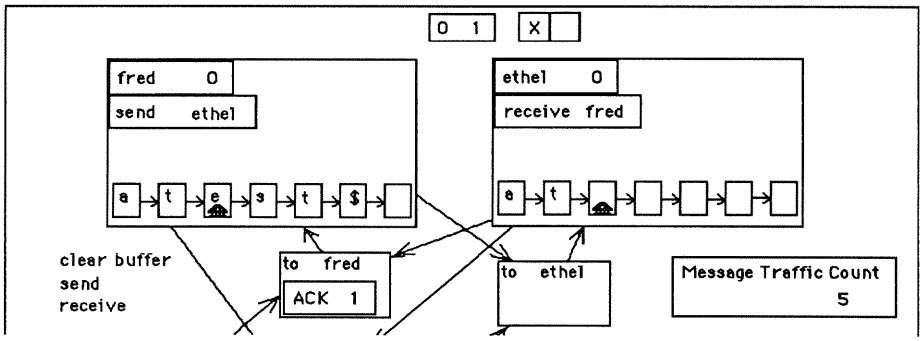


Figure 11 - Snapshot of protocol simulation

4. Second example: a sliding window protocol

The sliding window protocol and its graphical specification is only described briefly here.

4.1 Description of the protocol

Unlike the alternating bit protocol, the problem addressed by sliding window protocols is a possible difference in speeds between the sender and the receiver. Message packets include a sequence number, and the receiver contains a buffer known as a window that can accept a range of sequence numbers. Any message in the window will be accepted (regardless of whether it is received in the proper order), but the messages are relayed by the receiver to the user in the proper order. This is achieved by only passing those messages on to the user when they reside at the "bottom" of the window, that is, at the part of the window with the lowest sequence numbers. When a message at the bottom has been received, it is relayed to the user, the message is acknowledged (acknowledgements contain the sequence number of the message being acknowledged), and the bottom and top pointers are incremented, so that the next message in the sequence is at the bottom. The receiver's window has a maximum size, so that if a message is received whose sequence number is above the window, but the window can be legally expanded to hold it, then the "top" pointer is increased to allow the message to be received and buffered in the window. These changes to the top and bottom pointers effectively cause the window to "slide."

The sender contains a window, too. If there is a message to be sent, and there is still room in the sender's window (it also has a fixed maximum size), the message is sent and is added to the window at the top. When the sender receives an acknowledgment that falls within the window, it moves up the bottom pointer in the window so that it points to the sequence number one past the acknowledged message (i.e., all messages up to and including the acknowledged message are removed from the window). The reason we can do this is that the receiver only acknowledges messages in order, and therefore, if it has acknowledged one message, it must have acknowledged all previous messages, whether they have been received or not.

Transmission lines that lose data are handled in the following way. The sender may resend any message inside the window. (A more efficient approach is to only resend those messages in the window for which no acknowledgement has been received.) The receiver may at any time resend the previous acknowledgement.

4.2 Specification of a sliding window protocol using ChemTrains

Figure 12 gives a snapshot of a network running the sliding window protocol. The nature of the solution is similar to that of the alternating bit protocol, but there are a number of differences. The message buffers contain the sequence numbers which are sent with the message as part of the message packet. Within each buffer, there are two types of pointers.

The white pointers indicate message slots within the window. The black pointer indicates the next message to be sent. It can reside in the same space as a white pointer. Also of interest is the box containing the digits 1 through 7. This is used in some rules to distinguish between the sequence number and the message in a message packet. The sequence number will match an element of the box with the digits, a message will not.

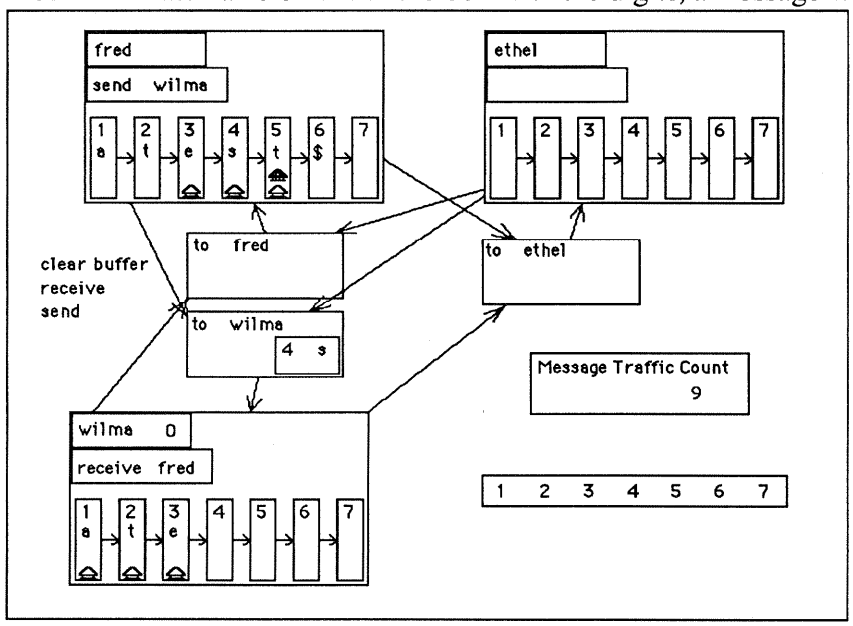


Figure 12 - Snapshot of the sliding window protocol

Two sample rules for the sliding window protocol are given in figures 13 and 14. The rule in figure 13 illustrates how the sender may run faster than the receiver. This rule, "send two messages," places two messages on the communications link. These two messages, denoted by the variables *x* and *y*, reside within the window, and *x* is the next message to be sent. The two messages are bundled with their sequence numbers, denoted by the variables *1* and *2*, and are placed on the appropriate link. As a result, the black pointer is moved up to point to the position past *y*, indicating that this next cell contains the next message to be sent. The window, however, cannot be moved until an acknowledgement is received, so the white pointers do not move.

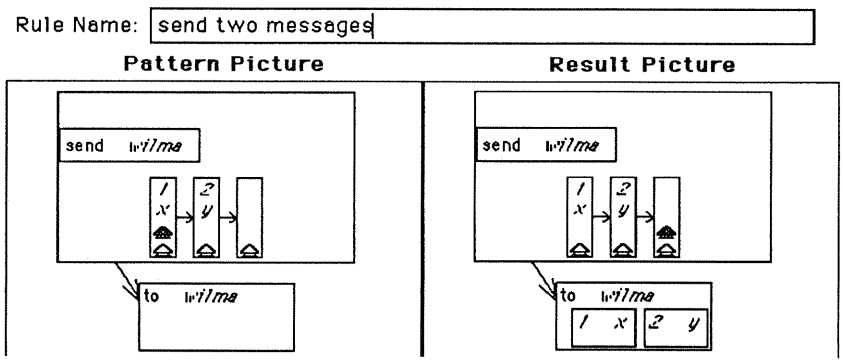


Figure 13 - Rule governing the sending of two messages

When an acknowledgement is received (figure 14), the pattern picture causes the buffer cell with the matching sequence number to be located. The result is that the entire window is moved up past the position of the acknowledged message.

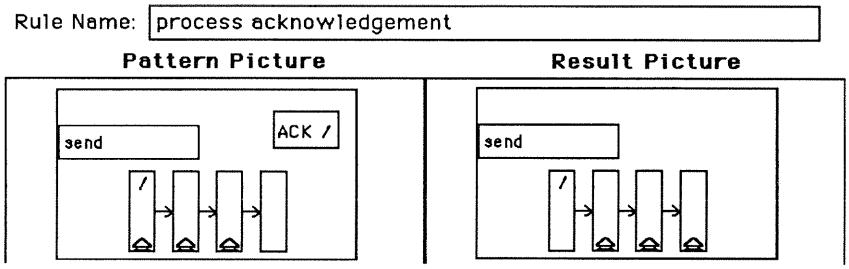


Figure 14 - Rule governing the receipt of an acknowledgement

5. Discussion

The solutions derived for the two protocol problems are straightforward to read and understand (although there were a few writability problems which will be discussed below). The rules are concise (although not as compact as textual formal specifications), and the functions of the individual rules were understandable simply through examination of the graphics; no knowledge of a textual specification method is necessary to understand them.

Although certain graphical idioms (such as the use of pointer objects and chains of boxes to represent message queues) were not immediately obvious to the programmer who implemented the sliding window protocol (a student with some experience in protocol design), once the idioms were demonstrated to him, they were readily understood and were easily transferable to other, related situations. The notion of top and bottom pointers in the sliding window protocol, for example, was derived directly from the message buffer solution in the ChemTrains version of the alternating bit protocol.

One advantageous aspect of using ChemTrains for protocol specification was the ease of modification of the specifications. Altering the simulated network configuration simply meant changing the picture that represented the initial network state. One could also easily change the characteristics of the communications media being simulated by adding, deleting, or changing rules corresponding to the behavior of the links. The behavior of the protocol on different types of links could also be evaluated by changing the firing probabilities for rules describing the links, thus making the links more or less noisy, for example. In addition, protocol behavior involving ambiguous specifications could be tested through the use of probabilistic rule firing, and changes to the probability of rule firings can be used to fine-tune the specification.

Finally, it is simple to add counters as graphical objects to rules, so that when a rule fires, the counter is incremented. This aids in performance evaluation, as we can count redundant messages, lost messages, time spent waiting for a message, and so on.

There were a number of problems that we encountered in attempting to specify the two protocols. The most prominent was that certain objects are simply not naturally graphical. Numbers and counters are examples of such objects. It is possible to implement counters graphically, and an early version of our solution did this, but later versions of the ChemTrains system incorporated counters as primitive objects in order to avoid this difficulty.

Another problem is that programmers may initially have trouble overcoming certain preconceptions inherited from using conventional programming methods. One example of this concerns the use of pointers. Conventionally, a pointer is a value that resides in a certain location and references some other object elsewhere. In ChemTrains solutions it is possible to represent pointers in this way, through the use of arrows actually pointing from one object (the pointer object) to another (the referenced object), but it is often more appropriate to dispense with the pointer object entirely and simply refer to objects by

placing graphical markers on them, then reference the object by using rules that require a match with that marker.

A third problem with the use of ChemTrains rules is that there is no way to specify negative conditions. This is unfortunate, since protocol rules may be of the form "If event X just happened, but event Y has not happened yet...," or "If event X has not happened within the last n seconds...." The latter, in particular, is especially common in specifying timeouts. To make a graphical system fully usable for specifications, such negative conditions must be expressible.

Another problem is that while it is simple to understand the operation of individual rules, it is often difficult to see how rules relate. This is a problem with rule-based systems in general, and is not inherently graphical. For example, in the sliding window protocol, consider the rule that states that receipt of an acknowledgement causes the bottom pointer to be moved up in the window past the acknowledged message. This rule might be puzzling to a reader of the specification unless he or she knew that the receiver only acknowledges messages in sequence order, and that the receipt of an acknowledgement implies that the receiver must have also acknowledged all the previous messages, even if those acknowledgements were never received. For improved readability, some method must be found for relating the functions of the rules.

The design goal of ChemTrains was to provide a language usable by people without computer science training, yet expressive enough to solve non-trivial problems. ChemTrains' usability has been demonstrated elsewhere [2,4], and its expressiveness is demonstrated by the work we have presented here. Criticisms from the two computer scientists specifying the protocols illustrate the tradeoff in the design of ChemTrains between making a language usable by non-programmers and by experienced programmers solving more complex problems in specialized domains. The fact that the computer scientists wanted features of modularization, typing, and abstraction that are found in other languages leads us to believe that to satisfy programmers working in specialized domains, it should be possible to be able to tailor the system or otherwise add higher-level overlays. (Such overlays in the protocol domain might include buffers, queues, messages, links, and generic communicating entities.) The low-level substrate, being more general, should still be made available to people who are not computer scientists.

6. Conclusion

We have demonstrated that simple communications protocols can easily be specified and simulated using graphical transformation rules. Such specifications and simulations are useful for the rapid prototyping of newly designed protocols (perhaps even before they are described formally), for gathering performance data on these high-level prototypes, and for experimenting with modifications to the protocols or to the environments on which they are intended to run. We believe that, like other algorithm animation applications, this technique will also be useful in illustrating the operation of specific protocols, and generally in the teaching of topics in telecommunications.

More ambitiously, we believe that it may be possible to generate prototype code that will run on the actual network hardware or even production code for the protocol, although this remains an open problem. These are issues we hope to pursue in future work.

Among other future work suggested by these issues are the problems discussed in the previous section. These are issues which must be addressed to allow more complex protocols to be expressed graphically. We expect that the answer to these problems is an overlay or extension to ChemTrains that would provide a set of graphical primitives modeling events and structures that occur in protocols and their specifications. In addition, an abstraction mechanism with parameterization should be developed. These are directions that we hope to pursue in the coming year.

Acknowledgements

The authors would like to thank John Rieman and anonymous reviewers for comments on a draft of this paper, David Honan for his work on the solution to the sliding window protocol problem, and Clayton Lewis and John Rieman for early ChemTrains design work.

References

1. Apple Computer, Inc. *Macintosh Allegro Common Lisp 1.3 Reference Manual,* 1989.
2. B. Bell, "Using Programming Walkthroughs to Design a Visual Language," Ph.D. dissertation, technical report CU-CS-581-92, Dept. of Computer Science, University of Colorado, Boulder, January 1992.
3. B. Bell, "ChemTrains: A Visual Language for Building Graphical Simulations," Technical report CU-CS-529-91, Dept. of Computer Science, University of Colorado, Boulder, October 1991.
4. B. Bell, Rieman, J., & Lewis, C., "Usability Testing of a Graphical Programming System: Things we missed in a programming walkthrough." *Proceedings of CHI'91*, pp. 7-13, 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. W. Citrin, "Simulation of Communications Architecture Specifications Using Prolog," To appear at 1992 ACM Symposium on Applied Computing, Kansas City.
7. 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.
8. A.A.R. Cockburn, "A Formalization of Temporal Message-flow Diagrams," *3rd International Symposium of Protocol Specification, Testing, and Verification*, Stockholm, June 1991.
9. M. Diaz and C. Vissers, "SEDOS: Designing Open Distributed Systems," *IEEE Software*, pp. 24-32, November 1989.
10. M. Diaz, *The Formal Description Technique Estelle*, North-Holland, Amsterdam, 1989.
11. G. Fischer, J. Grudin, A.C. Lemke, R. McCall, J. Ostwald, F. Shipman, and B. Reeves, "Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments," *Human-Computer Interaction Journal.* submitted.
12. C. Forgy, OPS5 User's Manual, Computer Science Department, Carnegie Mellon University, Pittsburgh, 1981. Technical Report CMU-CS-81-135.
13. G.W. Furnas, "New graphical reasoning models for understanding graphical interfaces," *Proceedings of CHI'91*, pp. 71-78, New Orleans, April 27-May 2, 1991.
14. K. M. Kahn and V. A. Saraswat, "Complete Visualizations of Concurrent Programs and Their Executions," *Proceedings 1990 IEEE Computer Society Workshop on Visual Languages*, pp. 7-15, Skokie, IL, October 1990.
15. W.C. Lynch, "Reliable Full-Duplex File Transmission over Half-Duplex Telephone Lines," *Communications of the ACM*, vol. 11:6, pp. 407-410, June 1968.
16. Mark Moriconi and Dwight F. Hare, "The PegaSys System: Pictures as Formal Documentation of Large Programs," *ACM Trans. on Prog. Langs. and Systems*, vol. 8:4, pp. 524-546, October 1986.
17. D. New and P. Amer, "Adding Graphics and Animation to Estelle," *Information and Software Technology*, vol. 32, no. 2, pp. 149-161, March 1990.
18. A. Newell, "Production Systems: Models of Control Structures," in*Visual Information Processing*, ed. W. Chase, pp. 463-526, Academic Press, New York, 1973.
19. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, 1981.