CARLA: A RULE LANGUAGE FOR SPECIFYING COMMUNICATIONS ARCHITECTURES

Wayne Citrin, Alistair Cockburn

CU-CS-742-94

CARLA: A RULE LANGUAGE FOR SPECIFYING COMMUNICATIONS ARCHITECTURES

Wayne Citrin, Alistair Cockburn

CU-CS-742-94 1994

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

Carla: A Rule Language for Specifying Communications Architectures

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

Alistair Cockburn
Humans and Technology, Inc.
Salt Lake City, UT

Abstract

Due to the unique requirements of a series of projects to specify communications architectures using graphical representations (Cara [12] and MFD [7]), we have developed the communications-oriented rule-based language Carla (**Ca**ra **R**ule **La**nguage), which provides an executable specification of the architecture being developed. Carla is designed to provide the ability to specify and simulate high-level, possibly incomplete, specifications of communications architectures, and to allow the developer to refine the specification through the addition of behavior-describing rules. Carla is also well-suited to creating black-box specifications of any system whose behavior depends on input/output history. We describe the features of the language, discuss various design issues, and provide examples of various communications protocols specified in Carla.

1.0 Introduction

Designing network protocols can be a difficult, time-consuming process. The goal of the network architect is to design an architecture meeting requirements for capability, speed, bandwidth, security, etc., while at the same time ensuring that the design is free of such undesirable properties as deadlock, livelock, and unspecified responses to certain inputs in certain states. At the same time, the design task should be completed within a reasonable amount of time. The goal of the Cara project [12] was to assist in this design task by providing the architect with a visual paradigm for the representation of network architectures, a methodology for developing architectures according to that paradigm, and an environment for supporting the method. As part of Cara, the system provides a textual, executable representation of the specified architecture, Carla (for **Ca**ra **R**ule **La**nguage). Carla is also being used in a subsequent project, MFD [7], which extends the Cara model.

Carla is a rule-based language with several novel features that distinguish it from other rule based languages, including OPS5 [15], in that it offers communications primitives and an explicit temporal component in the message history facility. These features make Carla suitable for specification not only of communications architectures, but of any history-sensitive system.

1.1 Motivations

The design of Carla was carried out with the following considerations in mind:

- A specification language should function at a high level, specifying *what* should happen, but avoiding as much as possible dictating *how* it should happen. In other words, the specification language should support *black-box* specifications whenever possible. At the same time, the specification should still be executable.
- A specification for communications architectures should model the actual network, which is a set of independent, parallel entities. The more conventional model of network architecture specification using a scheduler and a dispatcher is an artifact of older multiprogramming systems and does not contribute to an understanding of the architecture being specified.
- There should be a convenient mapping from the message-flow diagrams (the visual paradigm used in Cara and MFD) to the specification language.
- The specification should executable even when it is incomplete.
- The specification should be modular: modification of the specification of the behavior of one type of entity should not affect the specification of the behavior of any other type of entity.
- The specification language should be amenable to formal methods, including proofs of properties. (Actually, this is currently only a secondary consideration in that none of our work has been directed toward proving properties of rule-based specifications, but it is a desirable long-term goal.)

Since it is intended that the specifications be generated automatically by the Cara system, human readability and constructability were not major considerations, the examples in section 4 show that writing of Carla programs is not onerous, and in some cases is simpler than alternative methods.

Conventional specification methods for communication architecture specification, particularly extended finite-state machines (exemplified by the Estelle language [13]), and process algebra (exemplified by LOTOS [3]) are do not meet these requirements because they do not easily allow for simulation of incomplete specifications and their implementations do not permit simple on-the-fly modification of specifications in the process of execution.

The notion that a specification should avoid as much as possible describing how things happen, led to consideration of a non-imperative model for Carla, and Carla was designed with logic programming in mind. Specifically, the desire to model independent parallel entities led to a concurrent logic programming model, in this case Guarded Horn Clauses [GHC] [23], a committed-choice concurrent logic programming language (that is, a member of a class of languages in which several rule guards are evaluated in parallel, and the system commits to one rule whose guard has succeeded), which led us to the use of a language based on rules containing guard/action pairs.

Committed-choice logic programming languages lend themselves to a style of programming modeling independent concurrent communicating entities [20], similar to that offered by Actors [1]. In this model, a procedure (generally, a set of rules) is defined which describes the behavior of an entity type, and instances of entities of that type are created through calls to that procedure. The each of the rules of the procedure (except for the termination rules) is tail recursive, thereby modeling entity execution as repeated application of the rules. The entities communicate through streams; in this case through partially instantiated lists. When a reader of a stream attempts to read an uninstantiated portion of the list, its execution is suspended. When a writer to the stream instantiates that portion of the data structure, execution of the reader can resume.

Another influence on the design of the language was the use of message-flow diagrams as the visual paradigm of the Cara system. They will be described in more detail in the next section, but may be characterized as a set of event points (corresponding to rule firings) with enabling conditions (corresponding to rule guards) and actions (corresponding to rule actions). One of the goals of the Cara project was to derive rules automatically from the diagrams where possible, and to derive them with the assistance of the network architect in other cases. (MFD attempts to extend this model by making the diagrams unambiguous and completely machine interpretable.) To be able to provide textual language counterparts for the elements of the diagrams would simplify this task.

One of the goals of both the Cara and the MFD projects is to be able to extract meaning from, and simulate, the dialogues described by the message-flow diagrams as early in the design process as possible. The sets of rules derived at that point in the design process will necessarily be incomplete, and the rules themselves may be incomplete or written at a higher level than is usually required for simulation. It is important that Carla specifications be executable even when they are incomplete in this way, although intervention on the part of the user may be necessary in many ambiguous or incomplete cases so that the intention of the user becomes apparent.

Another consideration, also related to the underlying model, is that modifications to the behavior of a single type of entity should not directly affect the behavior of other types of entities, except in that entities of the two types may interact directly during the course of execution. In particular, this means that entities of various types should not share code, but should only interact through network communications encountered during execution. It also means that extra-communicative synchronization of the entities should not be allowed; all communication and synchronization must be performed through message exchanges as part of the functioning of the protocol.

One final consideration is that the specification should be amenable to formal proof methods, so that certain properties may be discovered automatically, thus assisting the architect in the specification process. Logic-programming languages are amenable to automatic manipulation and analysis, and we hope that our rule language may exhibit similar properties.

The result is that Carla programs model parallel communicating objects, since this is the physical analog of the entities being modeled. Unlike the objects described in [22], however, the actual implementation is centralized and sequential. (See section 7.2.)

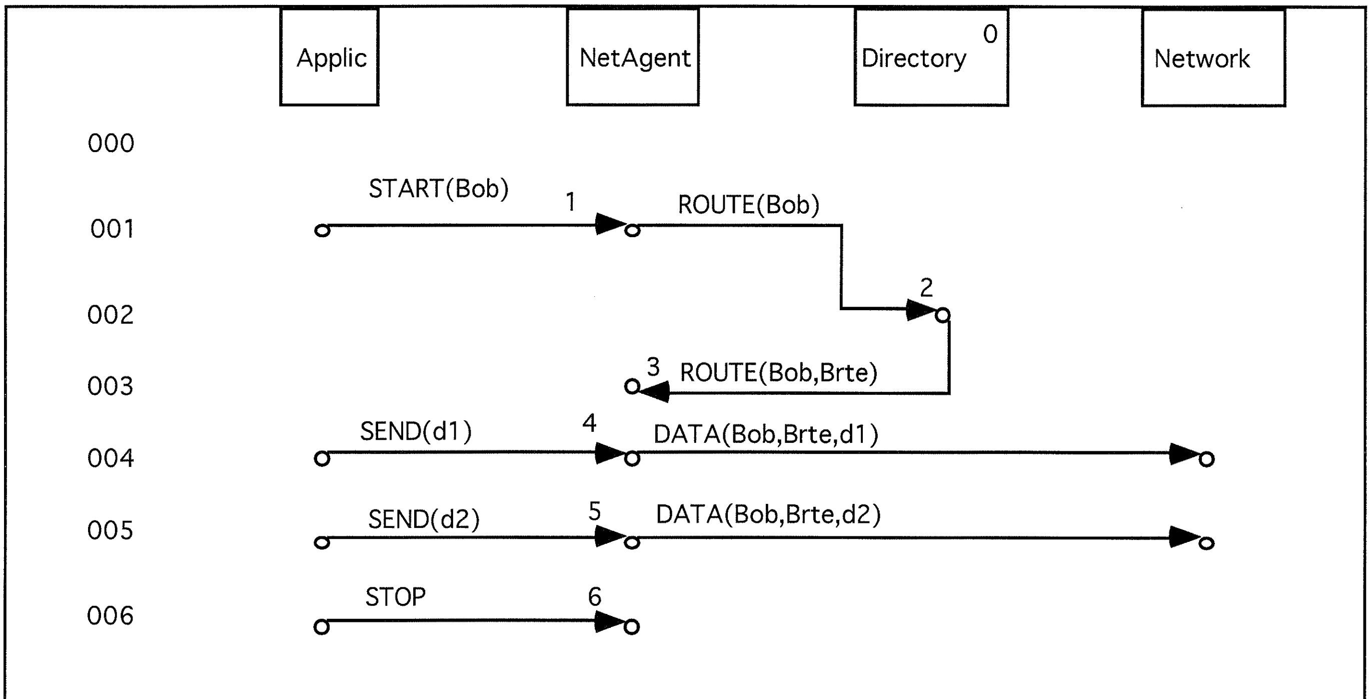
1.2 Message-flow diagrams

Message-flow diagrams, though informal, are a common method of describing a network architecture. The method is generally used to illustrate examples of dialogues allowed by the architecture, which is specified through other means. It is the thesis of the Cara and MFD projects that such message-flow diagrams can be formalized and used to at least partially specify a network architecture. The prevalence of such graphical representations suggests that they correspond to a certain extent to the designers' mental model of the problem being solved. Using such diagrams directly in the design process should lead to fewer errors and faster design.

The form of a message-flow diagram is illustrated by the example in figure 1. A message-flow diagram describes a dialogue between two or more participating entities (PEs). The PEs are denoted by a row of headers at the top of the diagram. Dots or circles in the space below a PE's header denote actions taken by that PE, and arrows denote messages passed from one PE to another. Time flows downward in the diagram within a PE, and an action point at the head of a message arrow (the receiving action) is considered to take place no earlier than the action point at the tail of the message arrow (the sending action). (In other words, messages cannot travel backward in time, although we do not rule out the possibility of instantaneous transmission of messages if the user chooses to define such transmission.) No other time relations are considered to exist between actions in the diagram.

In some cases, the interpretation of the diagrams is ambiguous. In such cases, footnotes are added to assist readers in understanding their meaning, as in figure 1 below, where a footnote indicates that the STOP message is used to end a conversation. [11] describes the formal meaning of message-flow diagrams, while

[10] describes aspects of machine interpretation of potentially ambiguous diagrams in Cara. [7] shows how diagrams are extended in MFD to make them unambiguously interpretable by machines.



Footnotes:

0) STATE: route(Bob,Brte)
1) START causes Route query
2) ROUTE req obtains route, if any
3) Route noted
4) SEND uses Route and Dst
5) Ditto
6) STOP stops communication

Figure 1. Simple message-flow diagram

Figure 1 is a scenario in which shows four PEs (Applic, NetAgent, Directory, and Network) engaging in a conversation. The conversation begins when Applic sends a START message to the network agent, indicating that it wants to 'talk' to Bob. The NetAgent consults the directory, first asking the directory PE for the route to Bob, and then getting the route Brte in return. Applic then sends several messages, which NetAgent packages with the destination (Bob) and the route to be used to get there (Brte). Finally, Applic sends a STOP message to end the conversation. The reader should note that there is nothing in the diagram itself to indicate that the conversation begins with a START message, that the route to Bob is indeed Brte, or that the conversation ends with a STOP message. These facts are added to the diagram with footnotes.

Message-flow diagrams may also be used to describe an entire class of dialogues. For example, variables can be added to messages, several instances of a message may be repeated (as in figure 1, where Applic sends two SEND messages; a typical interpretation of message-flow diagrams would allow an unlimited number of such messages before the STOP message is sent), and or several simultaneous, independent, and interleaved conversations may be conducted. We accommodate such multiply instantiated conversations through a concept called **context**; each independent conversation takes place in its own context, separate contexts are denoted graphically through different graphical styles (for example, colors), and the high-level concept of context subsumes much bookkeeping among different processes that would otherwise have to be specified [12]. Figure 2 shows multiply instantiated copies of the conversation described in figure 1, where the contexts are represented by solid or dashed lines.

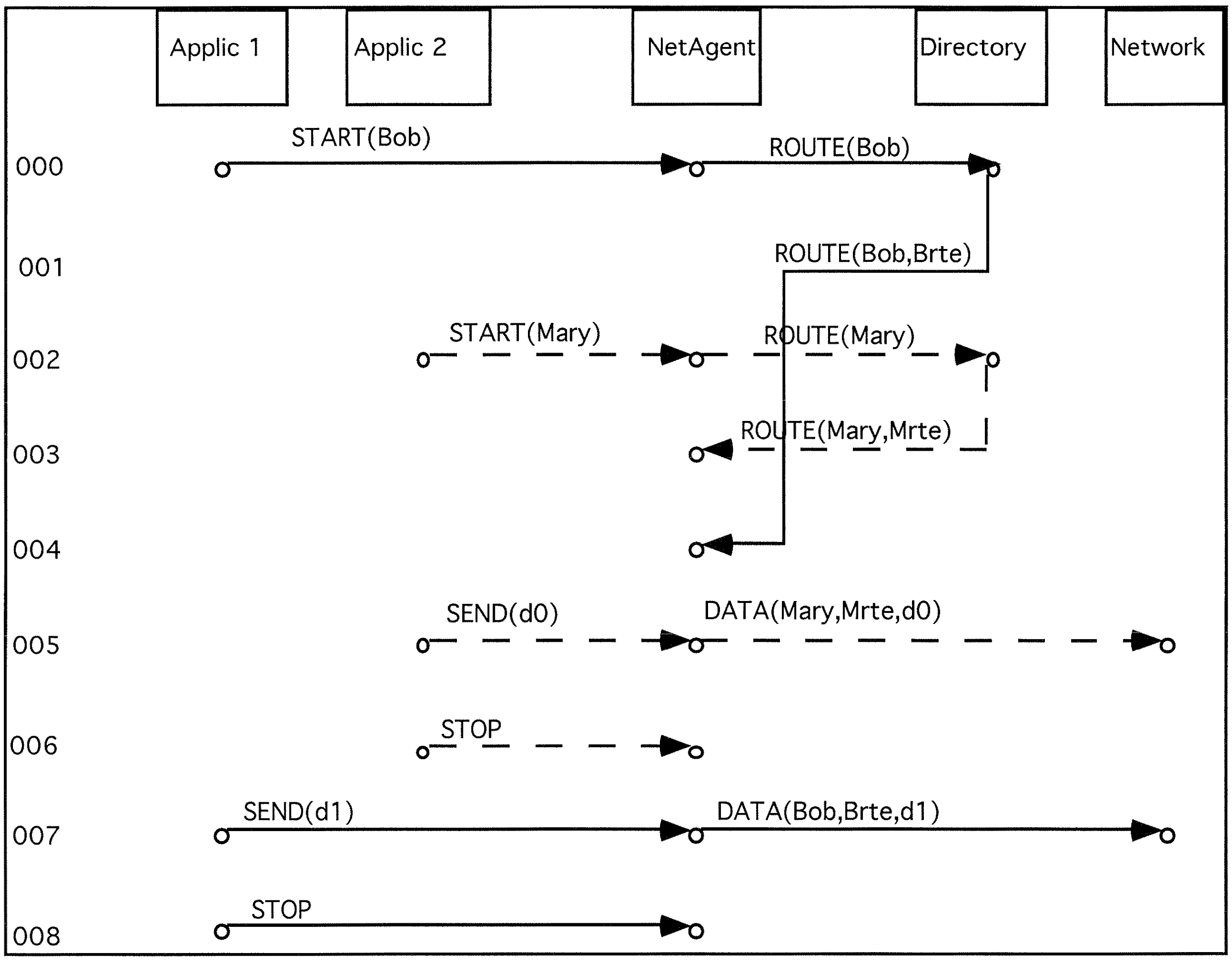


Figure 2. Message-flow diagram with multiple contexts

1.3 The Cara System and Carla

In the remainder of this paper, we motivate our discussion of the Carla language through reference to the Cara system. The newer MFD system will be described in a forthcoming paper.

The network architect uses Cara to specify an architecture by drawing a set of message-flow diagrams. The system attempts to deduce from the diagram, with the assistance of the user, rules governing the architecture's behavior, particularly the conditions under which an event point executes and the actions it performs. If the conditions in an uncompleted portion of the diagram (that is, a section of the diagram where no event points have been drawn) match rules previously extracted from that diagram or other diagrams, Cara attempts to complete the diagram (that is, continue the conversation between PEs) as far as its information will allow it. This facility can also be used to simulate already specified network architectures. The user can draw the initial conditions of a conversation, and Cara will use its stored information to act out the conversation in the form of a message-flow diagram. The user can then check whether the conversion between PEs has proceeded in an expected way.

Cara consists of three parts: the front-end **editor**, the central **translator**, and the back-end **simulator**. Using the editor, the user draws message-flow diagrams, which are transformed into a form understood by the translator. The translator extracts relevant data on architecture behavior, formulates them as Carla rules, and submits them to the simulator, which stores them in a database along with other facts concerning the diagram being drawn, specifically information concerning the state of the PEs and the messages transmitted between them. When Cara is simulating a conversation, or completing a diagram, the translator supplies the relevant conditions to the simulator, which examines its store of rules and state information and returns the next event occurring in the conversation. This information is passed by the translator back to the editor, which displays it graphically.

2.0 The Carla Execution Model

The execution model supported by Carla is a network of objects, called **participating entities**. (PEs) connected by **communications links** (also known as **commlinks**), along which messages travel. The PEs execute concurrently, receiving messages, altering their own state, and sending new messages, according to the behavior defined by their rules.

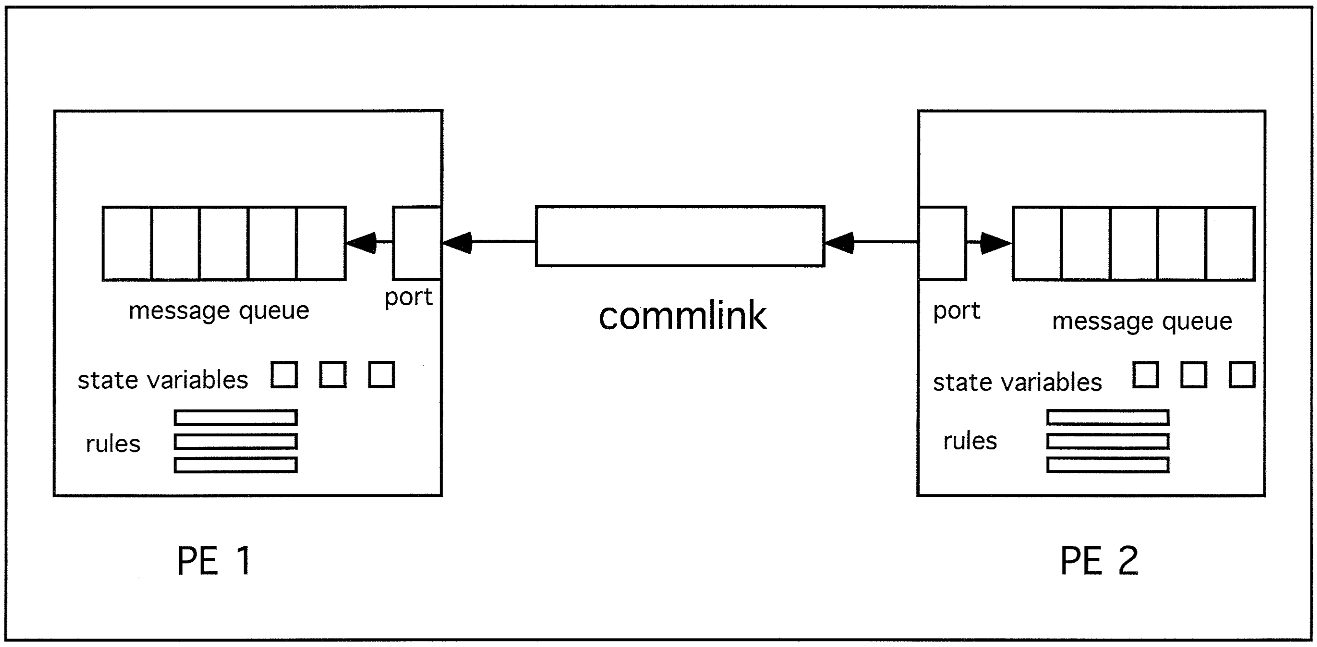


Figure 3. Carla execution model

The structure of a PE and its connection to the surrounding commlinks is given in figure 3. Each PE has zero or more **ports** through which messages may be sent and received. Some ports are designated **input** ports, and can be used to receive messages, and others are designated **output** ports and can be used to send messages. Associated with each port is a **history queue** in which previously received or sent messages are recorded for possible later reference. Each PE also contains a set of untyped state variables. There are also local variables associated with each activation of a rule. Each PE also contains a copy of the rules associated with its PE type. Each rule contains a guard that evaluates to true or false, and an associated action.

The actual behavior of commlinks is unspecified. They may represent queues, simple wires, or some other sort of medium. Transmission times may be delayed or instantaneous; messages may be transmitted in FIFO order or be reordered; the links may lose messages, corrupt them, or be perfect. In our simulator, link behavior is simulated through a separate link simulator, where the link is described through its own (possibly nondeterministic) rules of behavior, or manually by the user, who uses the graphic front end to convey messages from source to destination in the desired way. Communications links may have multiple sources and sinks, and each connection between source and sink may exhibit distinctly different behavior. The source end of a commlink simply accepts a value that is transmitted in some way to the sink ends. The sink ends of the commlink accept several commands. The **get** command requests that the commlink deposit a value of a given context (or of any context if the context value is left unspecified) in the PE's associated port. Depending on the availability of values and the behavior of the commlink, it might or might not then put a value in the PE's port. The command **take** tells the commlink that the PE has accepted or used the value. This information may be used in any way by the commlink, or be ignored entirely. For example, the link may be designed so that values that are **gotten** but not **taken** are saved for the next **get** command, while those that are **taken** are considered to be consumed and are now forgotten by the link. On the other hand, the link may be designed so that all **gotten** values are considered consumed and forgotten, regardless of whether they have been **taken**. In this way, a wide variety of link behavior can be modeled.

There is also a command **get_future**(n), which allows the PE to examine messages that will be received n **takes** in the future, if the commlink "knows" about them. Such a command is meaningful only for those types of commlinks that have buffers, and which have a value in the buffer that will be received n **takes** in the future. Values obtained by **get_future** are not actually placed in the ports, but rather in one of an arbitrary number of shadow ports associated with each port expressly for this task. **get_future** is necessary for description of the "future message history" facility described later in the paper.

Each PE independently performs its own **evaluation cycle** according to its own clock. At the beginning of the evaluation cycle, the PE's local clock "ticks" and a counter is incremented. For each input port in the PE, a **get** instruction is sent to the corresponding sink end of a commlink, and, depending on the link's behavior, a value might or might not be placed in the corresponding port. Next, the guards of all the PE's rules are simultaneously evaluated, and those which succeed are identified. If none succeed, the protocol specification is incomplete, and the user must intervene to modify the rule set and cover this new case. If more than one guard succeeds (that is, evaluates to **true**), the specification is ambiguous or nondeterministic. If the specification is nondeterministic, then the user may set the rule simulator to choose one executable rule and execute it. If the specification is not supposed to be nondeterministic, then the user may set the system to inform the user of multiple executable rules and allow the user to choose one of them to execute, or to modify the rule base so that the specification is no longer ambiguous. Once a single rule with a successful guard is decided upon, the PE **commits** to that rule, and for each port in the committed guard from which a variable was received, a **take** signal is sent to the corresponding end of the link, and the received value is recorded on the proper message history queue. Then the action part of the rule is executed (this may involve modifying state information or sending messages by placing them in output ports, and thence into a commlink), and the cycle is repeated.

3.0 The Carla Language

The following section is an elaboration of information presented in the previous chapter, viewed in the context of the Carla language rather than as an abstract execution model.

3.1 PE Types, Ports, and Variables

Each PE is an instance of a PE type, with which is associated internal structure (ports and variables) and behavior (rules). Each PE exhibits structure and behavior identical to other PEs in its type. Structural elements such as ports and variables are declared as parts of PE types, as are rules. When a PE is declared, it receives copies of the ports and variables associated with the type, and inherits the type's behavior.

PEs communicate with the outside world through **ports**. Sending a message corresponds to writing a value to a port, and receiving a message corresponds to reading a value from a port. Ports are further classified as input ports, through which messages can be received, and output ports, through which they can be sent. Attempts to use the wrong type of port will lead to an error.

Ports may have single or multiple instantiations. In the case of a singly instantiated port, only one instance of the port exists for any given PE, and it is simply referred to by name. Multiply instantiated ports have an indefinite number of instances in any given PE. They are analogous to an array of ports of indeterminate size. Multiply instantiated ports are referred to by a port name and a subscript value. If it does not matter on which port instance a message is received, the subscript value is left as an uninstantiated variable which is instantiated to the subscript value of the port instantiation on which the message is received. While *p1* may be an example of a singly instantiated port, a multiply instantiated port may be declared as *p1.**, and individual instantiations may have such names as *p1.1*, *p1.s3*, etc. A subscript may have any atomic value. Instantiations are named when a commlink is connected to a port, so that the set of meaningful instantiations of subscripts for a port corresponds to the set of port instantiations to which a commlink is connected. For example, if *p1.** is a multiply instantiated port belonging to PE type *T1*, and *PE1* is a PE of type *T1*, and for *PE1*'s port *p1*, commlinks are connected to *p1.1*, *p1.s3*, and *p1.out*, then *1*, *s3*, and *out* are the set of meaningful subscripts for *PE1*'s instantiation of *p1*. Other subscript values are not illegal, but no messages will be received through them, and no message sent through one of them will be received anywhere else.

Multiply instantiated ports are useful for cases where a message may arrive on any of a number of different ports, and the architect does not care on which one it actually arrives. For example, suppose a message may arrive at a PE on any of a number of ports named *in*. If *in* is a multiply instantiated port, and *in.S* (where *S* is an uninstantiated variable) is the port name operand of a **rcv** operation (see the section on **inter-entity communications**), then if a message is available on one of *in*'s ports, the message will be received and *S* will be instantiated to the subscript of the port through the message was received.

Any message received or sent through a port is automatically recorded on the history queue for the appropriate port.

Each PE type can have a list of variables associated with it. There are two types of variables: **local** variables and **state** variables. Local variables are associated with individual rules; new copies appear with each activation (execution) of a rule, and disappear when the activation ends. Each PE contains a copy if its type's state variables. These variables retain their values after a rule activation ends and the values are available for the next evaluation cycle. State variables may only be modified during the action part of a rule which is fired, using a special assignment operator, while local variables are considered logical variables whose values can be assigned once through unification at any point in the rule. (The next section gives more information on unification and logical variables and their role in rules.)

State variables must be explicitly declared as such. Any variable not explicitly declared to be a state variable is considered to be a variable local to the rule activation in which it appears.

3.2 Rules

3.2.1 Basic Concepts

The heart of the Carla language are the rules that govern the behavior of the PEs. Before presenting details of the rule syntax and semantics, we explain a few basic concepts:

The Logical Variable

The logical variable derives from Prolog and other logic programming languages [9], and ultimately from formal logic. It stands for a (possibly unspecified) entity, rather than for a storage location. Assertions concerning the equality of two logical variables are just that: assertions of equality, rather than actual assignments. For example, if we have two logical variables X and Y and say that X=Y, this is an assertion that X and Y refer to equal values, although we may not yet know what they are. Later, if we say that Y=2 (Y is equal to 2), it also must be the case that X=2. All local variables in Carla rules are logical variables.

Unification

There are several equality operations available in Carla, for example, the assignment of a value to a state variable, and the testing of equality between two values. However, the most common and versatile equality operation is the unification operation (denoted in Carla, as in Prolog, by the = operator). Unification is a symbolic operation which attempts to make its two operands identical by providing substitutes for the variables in the two operands. If successful, the result is to make the two operands identical, and since the logical variable is used, the substitutions also apply to all other occurrences of the variables. If unifying the two values is impossible (the values are said to be **ununifiable**), the operation fails. For example, a=b (unifying the two atoms a and b) will fail, as will f(X) = g(Y). (Capital letters here denote variables.) f(X,g(h)) = f(g(Z),g(Y)) is a unification that will succeed. One possible solution is to substitute g(Z) for X and h for Y. (Note that it is not necessary here to substitute anything for Z.) The result of the unification is f(g(Z),g(h)). If the two terms unify, there may be more than one possible substitution, or unifier, which will make them identical, but there is a most general unifier (**mgu**), which is unique up to the renaming of variables. All other unifiers are additional substitutions composed with (i.e., applied after) the substitutions of the mgu. When a unification operation appears in a Carla rule, it is assumed to mean unification using the mgu.

Because we wish to isolate the places at which a state variable may be modified, we place restrictions on the use of state variables in unification. Any attempt to modify the value of a state variable will cause the unification to fail. Also, if a local variable is unified with a state variable, and the value of the state variable is later changed, the value of the local variable remains as it was. We can consider the new and old values of the state variable to be different variables, and the local variable to have been unified with the old value.

3.2.2 Guards and Actions

Carla **rules** are of the form *guard* **->** *action*. This can be read as follows: if there is a solution to *guard*, find **one** such solution and perform the *action* using that solution.

The guard is a series of conjunctions and disjunctions, whose individual terms include message receptions, tests of message history, comparisons of values, and lookups of asserted facts. The terms can succeed or fail (in terms of the conjunctions and disjunctions, evaluate to **true** or **false** respectively) at any given time. In addition, if they succeed, they bind values to any uninstantiated variables in their arguments. These bound values do not necessarily need to be completely instantiated; they can be other uninstantiated variables. For example, the unification A=B succeeds when A and B are unbound variables before the unification. A term may succeed with more than one variable set of variable bindings, although only one solution is used at a time. For example, a term "the value X is received from port Y," where X and Y are variables, may have several solutions. Guards are evaluated in the Prolog manner [9], that is, from left to right with backtracking. When evaluation of a term fails, all variable bindings created by the evaluation of the previous term are undone, and execution of that previous term is re-attempted, in the hope of discovering a new solution. If there is a new solution, execution proceeds from there. Otherwise, execution backtracks to the previous term before that, variable bindings are undone, and the process is repeated. If backtracking returns to the beginning of the guard without having found an answer, the goal is considered to have failed. The order in which solutions are supplied upon backtracking is left as an implementation decision. This choice of ordering does not affect the truth or falsity of a guard, although it might affect the set of bindings chosen upon success.

Evaluation of all of a PE's rules' guards is considered to be performed in parallel, although, since guards do not have side effects until their rule is committed, they may be evaluated in any order.

The action is a (possibly empty) series of 'let' statements, followed by a series of assignment statements, message transmissions, and fact assertions. The 'let' statements are of the form

let(*local_var* = *expression*, ...)

and indicate that wherever the variable on the left-hand side of the statement occurs, it should be replaced by the evaluated expression on the right-hand side. 'Let' expressions are evaluated from left to right. All the statements in the action following the 'let' statements are considered to be performed in parallel, but, since they do not interfere with each other, may be performed in any order.

3.2.3 Inter-entity Communications

One of the principal stimuli for the firing of a rule is the receipt of a message from another PE. To express message receipt, Carla provides the **rcv** predicate. The form of **rcv** is

rcv(*port_name*,*value*,*context*)

Success of the **rcv** predicate means that a message *value* with the context *context* was found on the port *port_name*. Any or all of the above three arguments may be originally uninstantiated or incompletely instantiated, in which case they are unified with the relevant port, message, and context values when the **rcv** succeeds. An empty port (one in which no message has been received) is considered ununifiable with the argument *value*. If no port with a unifiable port name contains a unifiable value and context, the **rcv** predicate fails. Depending on the instantiation of the arguments and the values residing in the commlinks, **rcv** may have more than one solution. The order in which solutions are provided upon backtracking is an implementation decision.

If a context value is instantiated, only those messages of that context that are currently being held by the commlink connected to the port may be received. Depending on the properties of the associated commlink, it is possible for messages of the requested context to "jump the message queue" ahead of messages of other contexts that are ahead of it in the queue. (Recall that a **get** instruction to the commlink may take an associated context value that causes the commlink to supply only a message of the specified context, if one exists.) If the context value is uninstantiated, messages of any context may be received.

rcv may only appear in the guard of a rule. When a rule is committed, all commlinks connected to ports from which a message was received, are sent a **take** signal. Depending on the behavior of the commlink, this may cause the message to be removed from the commlink, it may be ignored, or the commlink may exhibit some other behavior.

The analog of the **rcv** predicate for transmitting messages is the **send** predicate:

send(*port_name*,*value*,*context*)

send may only appear in the action part of a rule. The action of **send** is to write a message with the given *value* and *context* to the port *port_name*. Once the message is written to the output port, the corresponding commlink processes it according to the commlink's behavior. *port_name* must be a fully instantiated value at the time the **send** is executed, which means that any variables in *port_name* must have been instantiated during the guard. *value* and *context* may be incompletely instantiated, in which case an incompletely instantiated message (i.e., a message with 'holes') is written to the port. If the message transmitted contains two occurrences of an uninstantiated variable, when the message is later received, these variables will still be unified, although they will not be unified with the variables in any other copies of the message that were sent to other PEs. This means that, if the message *m(X,X)* is sent by *PE1* to *PE2* and *PE3*, then *PE2* will essentially receive the message *m(X2,X2)* and *PE3* will receive the message *m(X3,X3)*, where *X*, *X2*, and *X3* are not the same variable. This decision was made to simplify implementation of Carla on our system, and it does not seem to reduce the utility of the system, since the purpose of being able to send incomplete messages was to allow the simulation of incompletely specified protocols, and not to allow communication between entities through the passing of logical variables, a possibility proposed by Shapiro and Takeuchi [20].

3.2.4 Contexts

A diagram in a specification typically describes a single conversation between PEs. However, many conversations ordinarily occur simultaneously over a network, and a single PE may itself be involved in a number of simultaneous conversations. It would be desirable to be able to simulate such behavior as a composite of several diagrams that have been previously drawn, particularly to learn whether there are any unintended interactions between the separate conversations. In order for Cara to interleave conversations, Carla provides the notion of **context**, which refers to a related, though not necessarily continuous, thread of conversation.

In the Cara system, a context is represented visually as the color or texture (i.e., dotted, solid, etc.) in which message arrows are drawn. Messages that are part of the same conversation are assigned the same context and displayed with the same color or texture.

In the underlying Carla system, contexts are represented by arbitrary values, such that there is a one-to-one correspondence between context values and graphical context attributes. Typically, the user assigns a context to the messages that initiate a conversation (either by textually assigning a context value, or, more likely, choosing graphical attributes for a message line), and rules of behavior are designed to receive messages in any context and send messages in the same context, although contexts are first-class values and may be manipulated in the same way as any other value. Similarly, rules may deal with multiple contexts.

Use of contexts allows the designer to avoid having to employ complex buffering or bookkeeping schemes to handle multiple conversations. At the early stages of development, the use of a high level construct like context is more appropriate. As a simple example of the use of contexts, consider a rule that receives two messages of the same context from ports *p1* and *p2*, adds the message values together, and sends the resulting message, in the same context, through port *p3*:

rcv(p1,M1,C), rcv(p2,M2,C) -> let(M3 = M1 + M2), send(p3,M3,C).

If there are two messages waiting in the commlink connected to *p1*: 1 (of context c1) and 2 (of context c2), and there is one message waiting in the commlink connected to *p2*: 3 (of context c1), then the result of firing the rule will be to send a message 4 of context c1 through port *p3*. If there were also a message 4 of context c2 on the commlink connected to port *p2*, the result of firing the rule would be either a message 4 (of context c1) or a message 6 (or context c2), depending on the properties of the commlinks.

3.2.5 History

Rule firings often do not only depend on events like message receptions occurring at the time of the firing. They may also depend on previous events, such as whether a certain message had been received or sent earlier. For example, if the messages *a*, *b*, and *c* should be sent out in that order, one could say that if *a* has been sent, then *b* should be sent, and if *a* and *b* have been sent, then *c* should be sent. The rules might also include a test as to whether the message one wishes to send has already been sent.

It may also be necessary to share data between event points. For example, in the message-flow diagram shown in figure 4, the PE receives a message *a(X)*, then at some later time receives a message *b*, at which point the message *c(X)* is transmitted (that is, the previously received value *X* is transmitted). By convention, the two variable identifiers X refer to the same variable. As a rule for the second event point, one should be able to say that if *a(X)* has been previously received (possibly on a certain port) and the message *b* is received, then the message *c(X)* is sent.

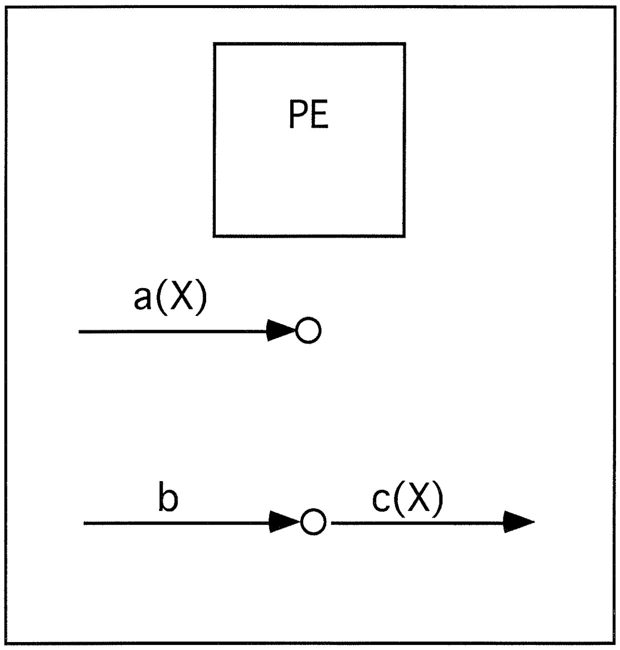


Figure 4. A diagram with message history

One way to implement these rules is through state variables. In the first example, there could be a variable State whose value is 0 if nothing has been sent, 1 if *a* has been sent but *b* and *c* have not been sent, 2 if *a* and *b* have been sent and *c* has not been sent, and 3 if all three have been sent out. Then the rules would look something like:

State = 0 -> send(out,*a*,Context), State := 1.
State = 1 -> send(out,*b*,Context), State := 2.
State = 2 -> send(out,*c*,Context), State := 3.

In the second example, we could use a state variable State to indicate whether or not *a* had been received, and another variable RcvdX to hold the received value of X. Those rules might look like:

State = 0, rcv(in,*a(X)*,Context) -> RcvdX := X, State := 1.
State = 1, rcv(in,*b*,Context) -> send(out,*c(RcvdX)*,Context).

Such a solution, however, runs into problems when several conversations like the one above, each with its own context, are running. It would not be possible to store received values in the presence of multiple contexts without resorting to more complex data structures or allowing state variable to have associated context (for example, to have a set of RcvdX indexed by context). If there were an upper bound on the number of possible contexts, one could store the received values in an array indexed by context, but such a choice would imply an implementation decision inappropriate to the high level of the specification.

In addition to avoiding context problems, the rules become much more readable when the message history can be explicitly referenced. It is not always clear what values the state variables reference, or even if they refer to message history.

The **message history** facility provided by Carla addresses these problems. Two operations are provided, both of which may be used in a rule guard:

> **rcvd**(*port*,*start*,±*occur*,*relative*,*absolute*,*message*,*context*)
> **sent**(*port*,*start*,±*occur*,*relative*,*absolute*,*message*,*context*)

A successful **rcvd** predicate can be read as follows: *message* with context *context* was received on *port relative* clock ticks ago, which happens to be *absolute* clock ticks after the creation of the PE. It was the *occur*th occurrence of the message (or a unifiable message in the case *message* is not initially fully instantiated) in the given context where the search started at the point *start* (representing an absolute clock value, 0 being the creation and the special value **now** referring to the current clock tick) and was done in the direction indicated by the sign of *occur* (where '+' refers to a search forward in time, and '-' refers to a search backward in time). An analogous interpretation is given to **sent**.

Any, all, or none of the arguments may be instantiated, and the order in which answers are returned upon backtracking is unspecified, except that if the *occurs* operand is uninstantiated, it is instantiated as negative, and if *start* is uninstantiated, it is instantiated as **now**. An *occur* value of -1 denotes the first matching item found (looking into the past), -2 the second item, etc.

Other conventions are that *relative* 0 refers to the current rule firing, and -1 refers to the previous rule firing, and so on. If a context is instantiated, the search will be restricted to the given context. If it is uninstantiated, each context will be examined in turn until a unifiable answer is found. If the context is instantiated to the special wild-card symbol '*', context in the history queue is disregarded, so that all elements on the queue are treated as though they were part of a single context.

Received values are only added to the history queue upon commitment, when a **take** message is sent to the commlink, so that messages are not recorded onto the queue unless they were actually received (and not just arrived at the port), and they are not visible until the next rule evaluation cycle.

Thus the first example given above could be written using message history as

```
not(sent(out,_,_,_,_,a,Context)) -> send(out,a,Context).

sent(out,_,_,_,_,a,Context), not(sent(out,_,_,_,_,b,Context)) -> send(out,b,Context).

sent(out,_,_,_,_,a,Context),
sent(out,_,_,_,_,b,Context),
not(sent(out,_,_,_,_,c,Context))
 -> send(out,c,Context).
```

This is perhaps more verbose than the state variable version, but certainly more explicit. Note that when a particular argument is a **don't care**, it can be replaced by the anonymous variable _, which unifies with anything. Also note that the comma between the terms in the guard (the **sent** and **not** operators) represents conjunction (the AND operator) while the semicolon (not shown here) represents disjunction (the OR operator). The **not** succeeds only when its argument, interpreted as a goal, fails, and fails when its argument succeeds.

Certain time relationships could be made more explicit in the above rules; for example, in the third rule we might want to make sure that *a* was sent before *b*, and in the second rule, we might only be interested in the situations when no *b* had been sent *after* the *a*. These relationships might be expressed by unifying variables with the respective *relative* arguments and comparing these results using a comparison statement. For example, the following two rules could express the time relationships added to the second and third rules listed above, respectively:

```
sent(out,_,_,_,AbsA,a,Context),
not((sent(out,_,_,_,AbsB,b,Context), AbsA < AbsB))
        -> send(out,b,Context).

sent(out,_,_,_,AbsA,a,Context),
sent(out,_,_,_,AbsB,b,Context),
AbsA < AbsB,
not(sent(out,_,_,_,_,c,Context))
        -> send(out,c,Context).
```

Unification and the logical variable can be used to retrieve and resend messages from the message history, even in the presence of arbitrary numbers of contexts (concurrent conversations) as shown in the following message-history version of the second example above:

```
rcv(in,a(X),Context) -> true.
rcv(in,b,Context), rcvd(in,now,-1,_,_,a(RcvdX),Context) -> send(out,c(RcvdX),Context).
```

The **rcvd** message history predicate only looks for those previously received messages in the same context as the message just received. The -1 value in the *occur* argument ensures that the most recently received message is retrieved, even if there are several intervening message receipts between the receipt of the *a*(*X*) and the *b* in different contexts. If a message was received between the *a*(*X*) and the *b* in the same context, the second rule will fail.

Message history is a powerful tool that can be used in another way. Some architectures require a sort of 'lookahead' on their communication links, modeled by buffering within the links, so that they will perform an action if the next message to be received (generally, the next message in the input queue or buffer) satisfies certain conditions. For example, we might want to say that if the next message to be received is an *a*, the PE should send out a *get_ready* message. Carla allows the architect to extend the message history facility to future message receptions, thus providing lookahead in the form of 'future history':

rcvd(in,**now**,+1,_,_,*a*,Context) -> send(out,*get_ready*,Context).

If the *occur* value is positive, it refers to future history. (An uninstantiated *occur* value is assumed to refer to past history.)

Future history is implemented in the Carla model using the **get_future** message sent to the commlink, and its availability depends on the characteristics of the commlink. It is only meaningful in association with those commlinks with some buffering capability, and implies that the PE is allowed to access those buffers. If that capability is not available, the future history reference will fail, and even if it is, the reference will fail if the future arrival of that message (is not known at the time the request was made. Future history is meaningless in the context of the **sent** operation, since future history represents reading the contents of a buffer whose contents have not yet been released by normal means. **rcv** actions may have this buffering, since it is possible that messages will arrive more quickly than the PE can process them, and it might be desirable to design commlinks so that those messages are not lost. In the case of **send** actions, however, the Cara execution model indicates that messages are transmitted over the commlinks as soon as they pass through output ports. Hence buffering is not needed.

3.2.6 Alarms

Communications architectures under development are by definition incomplete. It would be desirable to have a facility in the specification language to alert the architect that the simulated architecture is entering a section of the specification that has not yet been designed. Another possibility is that the simulated architecture is attempting something that should never happen; the system should report this situation to the architect, preferably with an informative error message. Finally, the architect may wish to supply sets of invariants to the system or to the rules. If these invariants are ever violated, the architect should be informed. In order to implement these constructs, an alarm facility is provided. The action **fail** is used to sound an alarm. **fail** takes one argument: a value which is returned to the user in the case the fail is executed. This value may be used to indicate to the user the cause of the failure. When the **fail** is executed, the associated value is transmitted to the user and execution suspends so that the user can intervene.

Thus, a user may, for example, at some future time intend to specify what occurs when a PE receives a message *a*. In the meantime, however, the user can add the rule rcv(in,*a*,Context) -> fail("Message *a* received").

which informs the user that an unanticipated message *a* was received.

Likewise, if only messages *a* and *b* should be received, the architect can add a rule to alert the user if some other value is received:

rcv(in,Msg,Context), not(Msg = *a*), not(Msg = *b*) -> fail("Invalid message received").

Finally, if there are invariants that must hold at the end of a rule, these invariants can be tested in through the addition of a rule that contains the inverse of the variant as part of the guard and a fail in the action. For example, if the variable Z must always be greater than 0 when an *a* is received, the rules

rcv(in,*a*,Context), Z > 0 -> <*do something*>.
rcv(in,*a*,Context), not(Z <= 0) -> fail("Invariant Z > 0 violated").

may be used.

It may be less wordy and more understandable to employ the **otherwise** guard in alarm rules. **otherwise** is a guard term that succeeds if and only if all the other rules belonging to the PE fail. For example, in the invariant example above, the second rule could be replaced by

rcv(in,*a*,Context), otherwise -> fail("Invariant Z > 0 violated").

In this case, the rcv terms in both the normal (non-alarm) rule and the alarm rule would succeed, but if the Z > 0 goal failed, then (assuming all the other rules controlling that PE's behavior also failed) the alarm rule would succeed. Likewise, in the second example, in which an illegal input is received, the alarm rule can be replaced by

rcv(in,Msg,Context), otherwise -> fail("Invalid message received").

The reception of a message that is neither *a* nor *b* will cause the above rule to succeed, assuming no other rule succeeds.

otherwise may also be used to catch so-far unspecified cases, as in the first example, but with a possible loss of information. For example, a general catchall rule

otherwise -> fail("Unanticipated situation").

could substitute for the rule in the first example. Such a rule is not too helpful, but will catch conditions for which no rule exists.

3.2.7 Fact assertions

One way in which a PE can keep track of state is to store it in state variables. However, another, more logical, method is to assert it. Such a representation may be more explicit and readable than somehow coding the information in a state variable. Again, like message history, asserted state is more general than state variables in that it remains valid even in the presence of arbitrarily large numbers of contexts. Another advantage of this method is that it may be considered a logical place holder for what in the future may be a subroutine call. For example, in the finished specification of a protocol, there could be a routine to compute the list of all PEs to which the given PE is connected. In early versions of the specification, however, this information could simply be asserted for the purpose of allowing early simulation of the protocol, then retrieved by the PE for the purpose of checking those assertions.

A state is asserted as an action. When it is asserted, it is associated with the PE that asserted it. Only one copy of a particular assertion may be associated with a PE at any one time. Different values denote different assertions, so that foo(1) and foo(2), for example, may coexist in one PE's state. If the user considers foo to be the state, and only wants one copy (for example, either foo(1) or foo(2), but not both), it is up to the user to write rules to enforce these user-defined restrictions.

When an asserted state appears as an action, the system checks whether the assertion is already present. If it is (i.e., if the assertion in the action unifies with an assertion in the PE's database), nothing is done. If the fact is not present, it is asserted. If the assertion action is surrounded by a **not** operator, the system removes the state if it is there (retracts it), and does nothing if it is not there.

When a fact appears as a guard, the system attempts to unify the fact in the guard with some fact in the database. This action may succeed or fail. The test can be embedded in a **not** operator, in which case the goal fails when the test succeeds, and vice versa.

Asserted fact checks are non-deterministic, and if more than one unifiable fact is found, one that causes the guard to succeed is chosen (through backtracking, if necessary).

As an example of the use of asserted facts, let us examine a specification in which a PE maintains and uses the aforementioned database of PEs to which it is connected. If a PE receives a message *connect(FromPE)*, it learns that it is connected to FromPE, and if it receives the message *disconnect(FromPE)* it learns that it is not connected to FromPE. In this example, it is not necessary that the PE be connected to FromPE to receive the disconnection message, or that it be disconnected from FromPE to receive the connection message. If a message *message(FromPE)* is received, and FromPE is not connected to the PE, an alarm is raised:

```
rcv(in,connect(FromPE),_)                                  -> connected(FromPE).
rcv(in,disconnect(FromPE),_)                               -> not(connected(FromPE)).
rcv(in,message(FromPE),_), connected(FromPE)               -> true.
rcv(in,message(FromPE),_), not(connected(FromPE))          -> fail("illegal message").
```

Note that **otherwise** may be used in place of the fact check in either guard. Also note that an assertion (action) or check (guard) is simply a structure appearing alone as a term. Any structure functor not reserved for another purpose (like **rcv**, **send**, **rcvd**, or **sent**) is interpreted in this situation as an assertion or check.

3.2.8 Other features

The preceding sections described the distinctive aspects of Carla. In this section, we will round out the description of the language by describing its more mundane aspects.

3.2.8.1 Expressions and types

The types provided by Carla are those provided by the underlying Prolog system. There are atoms, integers, floating-point numbers, structures, and strings. Variables may represent any type of value.

A number of operators are provided in Carla that may be composed into expressions, and that are evaluated in certain situations. There are the standard numeric operators (whose type casting and conversion rules depend on the rules of the underlying system): addition, subtraction, multiplication, division, integer remaindering, and bitwise AND, OR. and NOT. There are also operations for string concatenation, and substring extraction.

A **structure** is a compound operator, as in Prolog, consisting of a functor and a fixed number of arguments. (The number is defined by the **arity** of the functor.) For example, a(X,2,b(Y)) is a structure of arity 3 whose functor is **a**, and whose arguments are a variable, integer, and structure, respectively. Carla provides the Prolog operations **functor** and **arg** to extract or assign (depending on the action of the unification) the functor or arguments, respectively, of a structure.

3.2.8.2 Assignment to state variables

Carla allows two types of variables: local variables and state variables. Local variables are only active during an application of a rule, and behave as logical variables. (Among other things, they are single-assignment, and may be bound together.) State variables belong to the PE, and their values endure through repeated applications of rules. A state variable may be referred to by any or all the rules of the PE. Destructive assignment to state variables is allowed, during the action part of a committed rule, through the use of a special state variable assignment operator **:=**, which evaluates the expression on the right-hand side, and assigns the computed value to the state variable on the left-hand side.

Conceptually, all state variable assignments are performed simultaneously, and the results should be the same regardless of the actual order of execution. In order to assure this, each state variable is considered to have two values: the **old value**, which is the value of the state variable at the beginning of the rule evaluation cycle, and the **new value**, which is passed on to the next rule evaluation cycle. All references to a state variable on the right-hand side of an assignment statement refer to the old value, and all references to a state variable in the left-hand side of an assignment statement refer to the new value. All other references to a state variable anywhere else in a rule refer to the old value. There should be at most one assignment of a new value to a state variable in any given rule. If there is more than one, the result is undefined. If there is no assignment of a new value to a state variable, then at the end of the rule, the old value automatically becomes the new value.

4.0 Some Examples

4.1 A simple network conversation

Figures 1 and 2 illustrate a simple protocol in which an application requests a communication with a partner to which the route must first be obtained, sends some data, and ends the communication. The partner's name and route are sent with every data item. We note before going further that figures 1 and 2 do not completely specify the protocol. The relationship between succeeding STARTs and STOPs, in particular, is not illustrated. The rules in figure 5, on the other hand, do include that information.

It should also be noted that the Carla rules given in figures 5 and 6 are not intended to be written directly by a user, but rather by a diagram translation program such as that described in [10]. However, writing by hand the rules specifying both this protocol and the one in section 4.2 would not be too onerous, and use of Carla directly by users remains practical, particularly since the language's semantics and execution model are designed to facilitate protocol design. Minor refinements to the syntax, and development of an appropriate programming environment would address these problems.

We focus on the NetAgent entity. This simple protocol is of interest in that the NetAgent can serve multiple conversations operating concurrently, and needs to keep track of them separately. In the diagrams, as is appropriate for a specification at this level, no hint is given as to whether this ability to handle multiple conversations is implemented with a separate data structure or a separate process for each conversation. We are simply interested in an executable specification with the ability to handle multiple conversations, and create it by using the message context values to fully isolate the NetAgent's reaction to each message. A black-box specification of the NetAgent entity consisting of five Carla rules is shown in figure 6.

Rules 1 and 2 assert that a START is accepted only as the first message in the context, or if the most recent message in the context was a STOP. There is no stated bound on the number of instances of conversations the NetAgent can support. STARTs and STOPs of the different contexts do not interact at all. (Note that the rules do not specify what happens if the required order of START and STOP messages is violated. In such a case, no rule will have a guard that succeeds and execution of the program will halt.)

Rule 3 allows a route suggestion to be accepted as long as a route request was the last message sent in the context. No internal state variables are required to store the information for later use; it is accessed in rule 4 as part of the message history.

Rule 4 allows any number of data messages to be forwarded between the START and STOP messages. It says: a SEND message is to be accepted as long as the most recent route definition message came after the most recent route request, with no STOP occurring in the meantime. The three data items needed for the DATA message to be sent are identified: 'Dst' and 'Rte' are obtained from the message history, and 'Data' from the trigger message.

```
(1)  rcv('Applic'.N, 'START'(Dst), C1),
     not(rcvd('Applic'.N, 0, _, _, _, _, C1)),
     ->
     send('Directory', 'ROUTE'(Dst), C1)

(2)  rcv('Applic'.N, 'START'(Dst), C1),
     rcvd('Applic'.N, now, _, -1, _, 'STOP', C1)
     ->
     send('Directory', 'ROUTE'(Dst), C1)

(3)  rcv('Directory', 'ROUTE'(Dst,Rte), C1),
     sent('Directory', now, _, -1, _, 'ROUTE'(Dst), C1)
     ->
     true

(4)  rcv('Applic'.N, 'SEND'(Data), C1),
     sent('Directory', now, -1, _, QueryTime, 'ROUTE'(Dst), C1),
     rcvd('Directory', now, -1, _, AnsTime, 'ROUTE'(Dst,Rte), C1),
     AnsTime > QueryTime,
     not(rcvd('Applic'.N, now, -1, _, PrevStop, 'STOP', C1),
       PrevStop > QueryTime)
     ->
     send('Network', 'DATA'(Dst,Rte,Data), C1)

(5)  rcv('Applic'.N, 'STOP', C1)
     ->
     true
```

Figure 5. Rules for 'NetAgent' from figures 1 and 2.

Rule 5 indicates that a STOP may be accepted at any time. Again, no state variable is needed to note the arrival of the STOP. We return to this point in the discussion in section 6, on deriving code from specifications.

4.2 A sliding window protocol

We present a Carla version of the sliding window protocol originally given and verified in [14]. Since the protocol was originally presented using the Z notation, which easily separates into guards and actions, the behavior of the PE type 'Node' translates straightforwardly into eight Carla rules (figure 6). The specification in figure 6 makes use of message context and history, and eliminates the need for an internal message buffer as given in the original specification. Nonetheless, it is not a black-box specification, since it makes use of state variables ('S', 'R', 'A', 'E'). The example shows the flexibility of the Carla language, in that it can be used for black-box and for hybrid specifications, as appropriate.

In figure 6, the initialization rule INIT sets the initial values of the sent, retransmitted, acknowledged, and expected counters. The maximum window size is never explicitly set in this specification (nor was it in the original). It is assumed that some other mechanism defines that value (the user, for example). One simplification is made in this exposition: the sequence number 'S' is permitted to grow without bound, whereas in actuality it will wrap around at some maximum value.

Rule TRANSMIT governs the transmission of a data packet: the Node only accepts a SEND request as long as the number of outstanding data messages is less than the maximum window size. The data packet transmitted is given a sequence number one larger than 'S', the state variable used to keep track of the sequence numbers. In the Carla version, unlike the original and other published specifications, no internal state variable is required as message buffer, which, in the other versions, is used to store the messages for possible retransmission. Retransmission in this version makes direct use of history. This specification defines less of the internal structure of the Node, allowing implementors or compilers to use their own preferred methods to implement the message buffering.

The two RETRANSMIT rules govern retransmission: 'S' counts the number of packets sent, 'A' counts the number of packets acknowledged, so '$S-A > 0$' indicates there are acknowledgments outstanding. Whenever this inequality holds, the data packet associated with the retransmission number R may be resent. Two rules are needed to deal with the wraparound of R from S to A+1.

The RECEIVE rule states that only that data packet may be accepted whose sequence number matches the state variable 'E' (the expected sequence number). The data portion is forwarded to the user, and the expected value 'E' incremented.

Rule ACK allows an acknowledgment for all received packets to be sent at any time. Rule REC-ACK states that receipt of any acknowledgment constitutes acknowledgment of all packets with lower numbers. Redundant acknowledgments cause no damage.

```
(1: INIT)
        not(rcvd(_, now, _, _, _, _, _))
        -> S := 0, R := 0, A := 0, E := 1
(2: TRANSMIT)
        rcv('User', 'SEND'(Data), Cx),
        S-A < Window
        ->
        let( NewS = S+1, NewR = max(R,A+1)),
        send('Node', 'DATA'(NewS,Data), Cx),
        S := NewS, R := NewR
(3: RETRANSMIT)
        let(NextR = R+1),
        S-A > 0,
        R<S,
        sent('Node', now, -1, _, _, 'DATA'(NextR,Data), Cx)
        ->
        send('Node', 'DATA'(NextR,Data), Cx),
        R := NextR
(4: RETRANSMIT)
        let(NextR = A+1),
        S-A > 0,
        R=S,
        sent('Node', now, -1, _, _, 'DATA'(NextR,Data), Cx)
        ->
        send('Node', 'DATA'(NextR,Data), Cx),
        R := NextR
(5: RECEIVE)
        rcv('Node', 'DATA'(S,Data), Cx),
        S=E,
        ->
        let(NewE = E+1),
        send('User', 'RCVD'(Data), Cx),
        E := NewE
(6: REJECT)
        rcv('Node', 'DATA'(S,Data), Cx),
        S/=E,
        ->
        true
(7: ACK)
        true
        ->
        let(AckNum = E-1),
        send('Node', 'ACK'(AckNum), Cx)
(8: REC-ACK)
        rcv('Node', 'ACK'(AckNum), Cx)
        ->
        let(AckPlus1 = AckNum+1),
        A := max(A,AckNum),
        R := max(R,AckPlus1)
```

Figure 6. Sliding window protocol

The REJECT rule indicates that any data packet arriving out of order is received and ignored. In this rule we have chosen to receive the message and let it become part of the history of the port. An alternative would have been to define a new basic Carla action 'discard', which would take the data value from the port but not place it in the history of the port. This alternative, however, complicates the language model and leads to a difficulty in the interpretation of message histories on a system-wide basis: some messages are recorded as sent, but not recorded as either received or lost.

5.0 Design Issues

5.1 Influence of the message-flow diagrams

The use of message-flow diagrams as the Cara visual paradigm influenced the design of Carla in a number of ways. Foremost among these is the choice of a guard-action paradigm. The event points in the diagrams appear as dots with message arrows entering and/or exiting. The entering arrows may be seen as enabling conditions, and the exiting arrows may be seen as results. With the addition of further enabling conditions, such as numerical comparisons and fact and history checks, and other results including fact assertions and alterations to state variables, the mapping to guard-action pairs seems straightforward.

The concept of message history, as discussed earlier, reflects the influence of message-flow diagrams. Figure 4 gives an example of a message arrow that refers to a value appearing in a message arrow of a previous event point. Thus, there must be some method to refer to previous message traffic. As we have seen, this information could be encoded in the state of the PE, but such a practice yields rules that are more obscure than when the message history can be referenced directly.

Contexts were required to reflect the distinction between a diagram that defines the rules governing a single conversation, and composite diagrams that examine the consequences of multiple simultaneous instances of these and other conversations. A conversation is the analog of a process that covers several physically separate entities. In this case, the context functions as the conversation identifier. The combination of contexts and unification allows the PEs to keep conversations separate as part of the ordinary rule mechanism, without the need of special tables or a dispatching mechanism. This separation through context allows us to design a single conversation without worrying about how that conversation will be kept separate from other conversations. Conversations, of course, can still interact through state variables, but this provides a narrow, easily monitored window through which these interactions can be observed.

5.2 Influence of the Cara development methodology

The aims of the Cara development methodology influenced the design of the Carla language. One such aim was to provide incremental definition of communications architectures. It is intended that the architect be able to define several distinct aspects of the architecture simultaneously. In one sense, this means that the architect should be able to work on the definition of the behaviors of different PE types in parallel. The design of Carla suits this style of work, since the architect can define several PE types, and alternate between them in defining and refining their behavior. The PE types only interact through the messages their instances exchange, and a gap in the behavior of one type of PE due to an incomplete specification will not affect the defined behavior of another type of PE. The architect can add a rule to one PE type, then switch to a different PE type and add a rule to define that PE type's reaction to the newly defined behavior displayed by the first PE type.

Incremental definition also works within the specification of a single PE type. The rule-based paradigm makes it easy for the architect to define separate aspects of the behavior of a single PE type in parallel, since the rules defining the PE type's behavior do not interact except through the side effects of state variables and message history. The architect can progressively refine rules without altering rules already defined. If the simulation shows that the rules interact in an unintended way, they can be further refined.

Another way in which Cara influenced the development of Carla is the desire to provide execution or simulation of incomplete specifications. The architect should be allowed to execute *incomplete* rules, that is, rules with a guard but no action, or with an action but no guard. The system should also be able to simulate a specification when not all the rules are present. If a rule is missing, the system will indicate this to the user/architect; the system points out what is missing, and the user takes an action. Incomplete specifications also contain rules that are incompletely distinguished. In Carla, this takes the shape of multiple fireable rules: the user must provide assistance to the system and eventually refine the rules sufficiently that their firing conditions are distinct.

The mechanisms provided for executing incomplete specifications are user interaction, and logical variables and unification. The role of user interaction has just been described. Logical variables and unification allow the specification to manipulate messages symbolically, without worrying about their content. For example, the architect might be designing a protocol in which a PE of some type receives a message **go** and then transmits a message to a PE of a second type. When that PE receives the message, it relays it to a PE of a third type. Even if the contents or the format of the message have not yet been defined, the architect can still specify and execute what is known about the protocol. The receipt of the **go** and the transmission of the (so far unspecified) message may be expressed as

```
rcv(inPort,go,Context) -> send(outPort,Message,Context).
```

Note that a message is sent, but that so far, it is just a blank message. The relay rule may be expressed as

```
rcv(inPort,Message,Context) -> send(outPort,Message,Context).
```

Again, this rule works and is executable even if the message sent has an unspecified format. These unspecified messages can be manipulated and compared in more sophisticated ways. For example, instead of simply relaying the message every time it is received, let us assume that the PE only relays it if it receives the same message two consecutive times, but only if it hasn't already relayed the message the last time it received it. The first rule says that if a message is received and the last message received (in the same context) was not the same one, then the PE should do nothing:

```
rcv(inPort,M1,Context), rcvd(inPort,now,-1,_,_,M2,Context), M1 != M2 -> true.
```

The second rule says that if a message was received and the last message received was the same one, but that it already relayed the message upon the last receipt, then it should also do nothing:

```
rcv(inPort,Message,Context),
rcvd(inPort,now,-1,Abs,Rel,Message,Context),
sent(outPort,now,-1,Abs,Rel,Message,Context)
  -> true.
```

Finally, if the message was received, the same message was received last time, and but the message was not relayed last time, relay it now:

```
rcv(inPort,Message,Context),
rcvd(inPort,now,-1,Abs,Rel,Message,Context),
not(sent(outPort,now,-1,Abs,Rel,Message,Context))
  -> send(outPort,Message,Context).
```

Note that this specification 'works' (i.e., is executable) even when the messages are incomplete, and will continue to work when the rules governing the transmission of the original messages are refined so that the messages become completely specified. It should be noted that all uninstantiated messages will look the same to the rules given above, so if the architect wishes to distinguish between incompletely specified messages at this early stage, he or she will be forced to somehow specify them more completely in order to distinguish between them. For example, the dummy messages **m1** and **m2** may be used for the moment, or if the architect knows that two different types of message frames will be used, even if the format is not known, the incompletely specified messages **frame1(_)** and **frame2(_)** may be used. These frames can be more completely specified later during the process of refinement.

A final goal of Cara is that architectures should initially be specified and simulated on a high level and later progressively refined. In part, this means delaying the specification of certain low-level computations and inserting into the high-level specification enough information to allow the system to execute. In Carla, fact assertions can be used, at least at an early stage, in place of complex functions. For example, an architecture may contain a routing algorithm to determine the optimum route between two nodes given the starting and ending points. Early in the development of the architecture, however, the actual algorithm may not be important, and may not yet in fact have been written. The architect may still begin to specify and even execute the architecture, even if it employs routing information. Instead of having a routing routine, a number of facts representing routing information may be asserted as part of the initial conditions of the program. These facts will eventually be replaced by a routing algorithm that computes the message route. There is no reason, however, that the call to this routine should differ from the asserted fact check in the initial specification.

A rule that used the routing information to determine the next node to which to send the message might work as follows. Given the start and end nodes, the node would get the best route, and send the message to the first node on the node list (note that each node knows its own name, contained in the pseudo-variable **me**):

```
rcv(fromHost,m(Message,To),Context),
route(me,To,[Next|_]),
connected(Next,OutPort)
  -> send(OutPort,m(Message,To),Context).
```

Initially, **route** is a set of facts associating a list of nodes with a given starting and ending point. Next is the first element on that list, and corresponds to the next node on the path. **connected** is a set of facts associating PEs to which the current PE is connected with the port through which they may be reached. **route** and **connected** will eventually be replaced by routines to compute the route or maintain the connection table. As mentioned earlier, the calling interface to these routines may be identical to that to the facts, so that the rule itself need not be changed.

5.3 Communication architecture vs. simulation

Cara presents a programming-by-demonstration paradigm that also allows us to animate that which we have designed. We would like to keep the constant aspects of the specification (what we call the architecture) separate from information concerning any particular simulation.

The constant architecture information consists primarily of the rules, which determine the behavior of the PEs, but also of that information that imposes structure on the PEs. This structural information includes the PE types, which are used to group the architecture information, and the ports and state variables, which define the structure of a given PE type.

In order to simulate an architecture, a ‘network’ must be defined and set up. Such a network includes instances of PEs of the various types (we refer to these instances simply as ‘PEs’) and the ways in which they are connected. Thus, the static portion of a simulation includes declarations of PEs of the various types, commlinks, and specifications of which ports of which PEs are connected to which commlinks. The static portion also includes initial values for state variables, asserted facts, and message history. (It is possible that the beginning of a simulation will reflect the condition of a protocol in mid-execution; by selecting appropriate initial values for this information, we can simulate these conditions.) The dynamic portion of the simulation includes message traffic and changes in the values of state variables and asserted facts, along with the sequences of rules fired by each PE. In our implementation of the Cara simulator, we have been careful to keep simulation and architecture information separate. For more information on the organization and implementation of the Cara simulator, see [8]

6.0 Related Work

The design of Carla relates to several distinct lines of research in the specification of parallel programs used in communications architectures. Most important among these are rule-based languages and concurrent logic-programming languages.

The rule-based approach of most interest to us is the Communicating Rule Systems (CRS) proposed by Mackert and Neumeier-Mackert [17]. Their approach differs from ours in several key ways. They seem to have no provision for expressing conditions involving message history, and consequently their specification style is oriented toward encoded state. In fact, explicitly named states, like those found in ESTELLE [13], are an integral part of the language.

CRS also differs from Carla in the way in which rule firing is spread out over time. In CRS, a rule is fireable if it is startable (i.e., its starting condition holds), but its effects may be delayed to some later point in time, at which time the effects are carried out as an atomic action. The Carla approach (rule firing as an atomic action) seems more appropriate for mapping from message-flow diagrams and their associated event points.

A third difference is in the method by which the PE-equivalents in CRS (known as **rule system instances**) communicate. Communication is performed through **gate machines**, which are the equivalent of commlinks, but with more rigidly defined behavior. There are two protocols defined for gate machines: for synchronous and asynchronous communication, respectively. For actual execution of the specification that does not involve user intervention, this is probably a good idea, but for the exploratory simulations that we envision, involving extensive interaction with the user, manual simulation of the communications links, including cases in which the links are imperfect, seems appropriate.

We investigated committed-choice concurrent logic programming languages [19] as a programming model for Carla. We considered one particular concurrent logic programming language, Flat Guarded Horn Clauses (FGHC) [23] to be a good basis for a rule system for specifying communications architectures that was both symbolic and executable, allowed the parallel execution of multiple PEs, and imposed no order on the execution of the various components of the rules. There were two drawbacks to FGHC, however. The first was that the execution of FGHC rules did not allow for much user intervention. There is a great deal of nondeterminism in the execution of FGHC rules, and we wished to give the user the ability to explore specific behaviors. A new FGHC interpreter, however, could have been implemented allowing user intervention. The second drawback was more problematic. We wanted Carla to have the property that if a guard had at least one solution (i.e., if it succeeded), then the rule should succeed, and the variables in the guard should be instantiated to reflect the solution. This is especially important for guards having tests of message history or asserted facts. Carla allows the tests to be incompletely specified, and have the system, if it finds a matching solution, instantiate the variables accordingly. However, in FGHC it is very difficult to write such rules, particularly when the guard consists of several such generator terms (that is, terms that output a sequence of values one at a time upon backtracking) joined by a conjunction operator. This problem is solved in Carla by incorporating conventional Prolog backtracking semantics into the execution of Carla rule guards.

The notion of user-intervention in the simulation process to resolve cases of nondeterminism appears in the work of Briand *et al* [4], as part of the SINAPS LOTOS interpreter. The nature of the intervention, where all possible next actions are presented to the user for his or her choice, is quite similar to that of the Cara simulator.

While the details of the specification of communications protocols using Prolog is beyond the scope of this paper, we wish to take note of a number of projects where this was done. Besides the three described below, there are a number of others [4, 6, 21].

Von Bochmann *et al* [24] proposed the modeling of FSM state transition rules in Prolog. Prolog-based specifications of this type are suitable for testing whether a sequence of inputs is legal under the specified protocol, or for deriving the associated outputs. When properly constructed, it is also possible to generate valid sets of test inputs, although this may not be possible under the complexity of communications architectures in the real world. Unlike the Carla approach, direct specification in Prolog does not support the specification of incomplete protocols, nor the modification of the specification or the current simulation state during the simulation. It also does not seem to be possible to refer explicitly do previous input and output history.

Logrippo *et al* [16] also described the use of Prolog to specify a protocol using a state-based approach. They also noted the advantages of using Prolog specifications as both "recognizers" and "generators." Several of the problems described above also apply here. Significantly, they also noted the shortcomings of Prolog as a specification language due to its simplicity and generality of syntax, and pointed out that various domain-specific primitives would have to be added before Prolog was completely suitable as a communications specification language.

Pappalardo [18] took a somewhat different approach to the use of Prolog. He translated ECCS, a process algebra-based language similar to LOTOS, into Prolog. Although Pappalardo's specification method, like Logrippo's and von Bochmann's, would not allow for exploratory simulation in the form of on-the-fly alterations to the specification and simulation state during the simulation, it allows the possibility of nondeterministic execution, since the translated Prolog code automatically maintains choice points for use in backtracking. Because of the committed-choice nature of Carla, maintaining choice points and performing backtracking *between* rules (as opposed to doing it *within* rules, as Carla does it), would require extra support in the simulation environment.

7.0 Discussion

7.1 Contributions

Carla provides a number of features that are novel or otherwise of interest in the context of specification methods for communications architectures.

High-level execution

One of the bases of the Cara project is the notion that the specification should be simulated at as early a stage in its development as possible. One way in which Carla accommodates that goal is through the notion of high-level execution, particularly through the use of asserted facts. By using facts to simulate more complex routines, which may not yet have been designed or written, the architect can examine the consequences of selected high-level results of routines without actually having to design the routine itself.

Execution of incomplete programs

A second way in which Carla supports the goal of early simulation is through the ability to execute, or rather to simulate, incomplete programs. Missing rules are filled in through user intervention, as is non-determinism due to incompletely specified guards. In addition, simulation may involve the passing and manipulation of incompletely specified data structures through the use of the logical variable. These variables represent values that may or may not have been defined, but which may still be manipulated. Other rule-based approaches like CRS [17], require specifications to be complete before they may be simulated.

Invariants

Verification using invariants may be performed through the use of rule guards and explicit failure. Rules may have checks for the truth of the invariant added to their guards, or special rules may be created simply to check the violation of invariants and raise an alarm. Invariants are explicitly found in UNITY [5], a nonexecutable specification method, and may be implemented in other rule-based systems like CRS in a manner similar to that of Carla, although the designers of CRS have not pointed this out.

Abstraction of communications media behavior

Carla does not require any pre-simulation specification of the behavior of the communications links. In real life, most communications channels exhibit non-deterministic behavior (e.g., they lose or corrupt data, or may reorder data). In addition, when hierarchical specification of communications architectures is used, as is generally the case, then to any given layer, functions of the lower layers may be viewed as properties of the communications medium. For example, if a layer requires correct, properly ordered data, then the communications medium which it sees will look perfect. To the lower level that actually handles communications errors, the channel will look like an imperfect medium. Thus, the same channel will have different properties depending on what part of the architecture is looking at it. We therefore decided that, since the architect may wish to simulate the architecture on a particular communications behavior (e.g., what would happen if a particular delay occurred), it would be best to let him or her actually simulate that behavior by hand at simulation time. It may still be desirable to allow the architect to specify the boundaries of legal behavior of a communications link, and inform the user when the desired behavior lies outside the legal boundaries. The section on future work discusses this point further.

CRS coordinates communications between **inference machines** (their equivalent to PEs) through the use of **gate machines**, which handle event requests between inference machines through either a synchronous or asynchronous protocol. We chose not to use this approach first because the interface between the gate machine and the inference machine is more complex than that between the commlink and the PE, and second because the range of behaviors of the commlink is more general. The advantage of the gate machine is that its fixed behavior allows the rules to be executable without intervention, while in Carla the simulation must either be accompanied by some user intervention or through the use of a scheduler module in the simulator.

Context versus Processes

We chose not to use a process-oriented model of execution in Carla, since such an approach is generally associated with a single machine, and it seemed to us that associations between events on separate machines were equally significant. The uses of contexts provides an elegant way of associating separate events that are part of the same conversation. Carla allows the architect to manipulate contexts if he or she desires it, but if a rule involves only one context at a time, simply using a single context variable in all places where message traffic is referred to will serve to keep conversations separate.

The process model can also be distributed over several machines, but it requires some kind of special communication between execution units. This special communication can either be associated with the messages, in which case it reduces to the context model, or it can be associated with some sort of external dispatcher. Because we wish to avoid any communication between PEs outside of the messages exchanged between PEs (in the real world, physically separated PEs would not have any such extra-communicative connection), we avoid the notion of a dispatcher. Conversely, however, if we wish to simulate PEs controlled through a dispatcher, we could do this through message passing.

History versus State

Finite state machines are a ubiquitous method of specifying communications protocols. However, the information encoded in the state often obscures the actual sequence of events that led to that state. Conversely, encoding that sequence of events can be a non-trivial process, especially when one wishes to minimize the number of states. It makes more sense to refer to the event sequence (in this case, the message history) directly. Such specifications are simpler to read and write. We have provided a set of primitives for expressing message history that appears to be sufficiently rich.

While state and history are functionally equivalent, the presence of multiple contexts makes the encoding of all but simple information in state variables problematic, as was shown in section 3.2.5. Multiple contexts makes it necessary to index state information by context. If the number of possible contexts is bounded, it is possible to do this indexing with arrays, but this represents an implementation decision that should not be made at this point in the design process.

A further value of use of message history comes during the simulation of the specification. Cara allows the architect to alter the simulation as it is proceeding, and even to alter events that have already taken place. For example, the architect can alter the previous message history in order to see what might happen if the past were different, or he or she might wish to set up certain initial conditions involving a postulated message history that might have existed before the simulation proper started. Such alterations and initializations are a simple matter, involving changes and initialization of the message history queues. In a finite state machine, choosing a state reflecting the new or preinitialized message history would be far more difficult, and such a state might not even exist.

Future history versus Queues

As part of the abstraction of the behavior of the commlinks as it appears to the PEs, we have created the idea of future history. The method by which a commlink may know about the future is of no concern to the PE, and may involve queues, buffers, randomness, omniscience, or some other method. Some commlinks may not be able to offer future history. There seems to be no inherent distinction between past message history and future history, and we feel that the ability to use future history is a powerful new tool for specifying communications architectures.

Lack of explicit scheduling

The Carla model, involving multiple asynchronous PEs, allows us to avoid explicit scheduling of PE execution. Any synchronization in the architecture is purely a feature of the specified communications between the PEs. This, plus the context feature, allows us to avoid an explicit scheduler/dispatcher. Avoiding a dispatcher is an advantage in protocol simulation, since there are many possible interleavings of rule firings between the PEs, and certain of these may be of particular interest to the architect. The Cara simulator informs the user which PEs have fireable rules at some given point in time, and employs user intervention to allow the architect to fire as many or as few of those rules as he or she wishes, thus allowing the simulation of any valid interleaving.

7.2 Implementation

Carla rules may be interpreted as inferences on a set of facts defining the previous message traffic and state values of a PE, and specifying new facts (namely, new messages and new state) that follow from it. Using this model, the Cara simulator implements a forward-chaining rule-based system. Message transmissions and receptions, changes in state variables and asserted facts, all associated with a simulation time, are all part of a **simulation database**. There is no distinction made in the simulation database between those events that happened in the past, and those that will happen in the future. The user chooses a PE or PEs of interest, and a simulation time of interest, and the simulator surveys the **rule database** to determine those rules associated with the PE (or rather its PE type) that are valid inferences (i.e., the guards are true), and alters the database by adding new facts to take account of the inference. Such a scheme allows the user to easily alter the database to try new hypothetical or alternative situations, or to set up new initial conditions and observe their effect on the architecture as defined by the rules.

See [8] for more information of the implementation of the Cara simulator.

7.3 Further work

Our work suggests several new research directions, as well as further refinements. One direction of work is to add more to Carla and the simulator to support the refinement of complex diagrams. In our system PEs are flat entities, but message-flow diagrams often include PE hierarchies. An event within one PE may actually involve a complex conversation between sub-PEs before any action is visible from the main PE. Right now, the best we can offer is to define and simulate the behavior of all the lowest-level PE and have the Cara editor abstract the higher-level information before it is presented to the user. Some method of defining PE type hierarchies, and specifying the behavior at various levels of abstraction (and making sure that the behaviors on each level are consistent) would be desirable.

A related problem is the modeling of network architecture stacks. Addressing the problem of architecture stacks should somehow involve the hierarchical levels of abstraction proposed above.

Finally, we would like to produce a formal description of the semantics of Carla rules. It seems reasonable to assume that such a description would involve a form of temporal logic concerning message sequences and changes of state. Such a description would help us develop formal methods, such as proofs of various properties, in a Carla specification.

7.4 Current status

The Cara system, including Carla, has been implemented, and has been used to specify some relatively small protocols [10]. The editor module of Cara has also been modified to allow the preparation of documentation involving message-flow diagrams. Based on our experiences with Cara and the use of inference and user interaction to eliminate ambiguity in the specification, work is proceeding on another language, MFD [2, 7], which allows the user to explicitly specify causality through the use of additional graphical notation. For example, history connections are indicated explicitly. The objective is to allow the user to specify protocols by demonstration, but at the same time make the system's task easier by making the designer's intentions explicit, something that is not usually done in programming by demonstration systems. One interesting feature of the language is that its design was refined through the use of a programming walkthrough [2], an iterative process in which prospective users attempt to solve problems in the prototype language, their difficulties are addressed, and the subsequently modified language is subjected to another round of testing. One important byproduct of the process is an understanding of the problems encountered by a programmer in that particular domain, and the ways in which a language may be used to address those problems. Both the Carla language and its simulator are part of the MFD environment currently being constructed.

Acknowledgments

This work was performed by the authors while both were at the IBM Zürich Research Laboratory. The work benefited from many constructive conversations with the other members of the Cara project and the Communications Architecture Software and Technology group there. We would particularly like to thank Rainer Hauser, Liba Svobodova, and Willibald Doeringer for their insights and suggestions. Thanks should also go to Benjamin Zorn of the University of Colorado for his constructive comments on the paper.

Work on the MFD language is funded by the National Science Foundation under grant CCR-9208486.

References

[1] Agha, G., *A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press; 1986.

[2] Bell, B., W. Citrin, C. Lewis, *et al.*, "The Programming Walkthrough: A Structured Method for Assessing the Writability of Programming Languages." *Software: Practice and Experience*, **24**: 1-25; 1994.

[3] Bolognesi, T. and E. Brinksma, "Introduction to the ISO Specification Language LOTOS." *Computer Networks and ISDN Systems*, **14**: 25-59; 1987.

[4] Briand, J. P., M. C. Fehri, L. Logrippo, *et al.*, "Executing LOTOS Specifications," in *Protocol Specification, Testing, and Verification, VI,* Sarikaya, B. and G. von Bochmann, eds. Amsterdam: North-Holland; 1987.

[5] Chandy, K. and J. Misra, *Parallel Program Design: A Foundation*. Reading, MA: Addison-Wesley; 1988.

[6] Choquet, N., L. Fribourg, and A. Mauboussin, "Runnable Protocol Specifications Using the Logic Interpreter SLOG," in *Protocol Specification, Testing, and Verification, V,* Diaz, M., ed. Elsevier Science Publishers B.V. (North-Holland); 1987.

[7] Citrin, W., "Design Considerations for a Visual Language for Communications Architecture Specifications," in *IEEE Workshop on Visual Languages*. Kobe, Japan, 1991.

[8] Citrin, W., "Simulation of Communications Architecture Specifications Using Prolog." *Applied Computing Review*, **1**: 10-17; 1993.

[9] Clocksin, W. and C. Mellish, *Programming in Prolog*. Berlin: Springer-Verlag; 1981.

[10] Cockburn, A., W. Citrin, R. F. Hauser, *et al.*, "Using Formalized Temporal Message-flow Diagrams," IBM Research Report RZ 2497, IBM Zürich Research Laboratory, 1993.

[11] Cockburn, A. A. R., "A Formalization of Temporal Message-flow Diagrams," in *Protocol Specification, Testing, and Verification, XI,* Jonsson, B., J. Parrow, and B. Pehrson, eds. Amsterdam: North-Holland; 1991.

[12] Cockburn, A. A. R., W. Citrin, R. F. Hauser, *et al.*, "An Environment for Interactive Design of Communications Architectures," in *Protocol Specification, Testing, and Verification, X,* Logrippo, L., R. L. Probert, and H. Ural, eds. Amsterdam: North-Holland; 1990.

[13] Diaz, M., *The Formal Description Technique Estelle*. Amsterdam: North-Holland; 1989.

[14] Duke, R., I. Hayes, P. King, *et al.*, "Protocol specification and verification using Z," in *Protocol Specification, Testing, and Verification, VIII*. Amsterdam, 1988.

[15] Forgy, C., *OPS5 User's Manual*. Pittsburgh: Computer Science Department, Carnegie Mellon University; 1981.

[16] Logrippo, L., D. Simon, and H. Ural, "Executable Description of the OSI Transport Service in Prolog," in *Protocol Specification, Testing, and Verification, IV,* Yemeni, Y., R. Strom, and S. Yemeni, eds. Elsevier Science Publishers B.V. (North-Holland); 1985 .

[17] Mackert, L. F. and I. B. Neumeier-Mackert, "Communicating Rule Systems," in *Protocol Specification, Testing, and Verification, VII,* Rudin, H. and C. H. West, eds. Elsevier Science Publishers B.V. (North-Holland); 1987.

[18] Pappalardo, G., "Experiences with a Verification and Simulation Tool for Behavioral Languages," in *Protocol Specification, Testing, and Verification, VII,* Rudin, H. and C. H. West, eds. Amsterdam: Elsevier Science Publishers B.V. (North-Holland); 1987.

[19] Shapiro, E., ed. *Concurrent Prolog: Collected Papers.* Vol. 2., Cambridge, MA: MIT Press, 1987.

[20] Shapiro, E. and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," in *Concurrent Prolog: Collected Papers,* Shapiro, E., ed. Cambridge, MA: MIT Press; 1987.

[21] Sidhu, D. P., "Protocol Verification via Executable Logic Specifications," in *Protocol Specification, Testing, and Verification, III,* Rudin, H. and C. West, eds. Elsevier Science Publishers B.V. (North-Holland); 1983 .

[22] Tanenbaum, A. S., M. F. Kaashoek, and H. E. Bal, "Parallel Programming Using Shared Objects and Broadcasting." *Computer*, **25**: 10-19; 1992.

[23] Ueda, K., "Guarded Horn Clauses," in *Concurrent Prolog: Collected Papers,* Shapiro, E., ed. Cambridge, MA: MIT Press; 1987.

[24] von Bochmann, G., R. Dssouli, W. Lopes de Souza, *et al.*, "Use of Prolog for Building Protocol Design Tools," in *Protocol Specification, Testing, and Verification, V,* Diaz, M., ed. Elsevier Science Publishers B.V. (North-Holland); 1986.