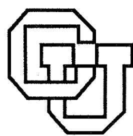
Visualization-Based Visual Programming Specifications Using Prolog

Wayne Citrin

CU-CS-543-91 September 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Visualization-Based Visual Programming Specifications Using Prolog

Wayne Citrin

CU-CS-543-91

September 1991

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

citrin@soglio.colorado.edu

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

Submitted to the 1992 ACM Symposium on Applied Computing

Simulation of Communications Architecture Specifications Using Prolog

Wayne Citrin

Department of Electrical and Computer Engineering
University of Colorado, Boulder

ABSTRACT

The design of the Cara simulator, a Prolog-based simulator for communications architecture specifications, is described. Unlike other Prolog-based simulation methods, the Cara simulator supports "exploratory simulation," in which high-level, incomplete specifications may be simulated, and various specification alternatives and elaborations added to the specification during the course of the simulation. Unlike other simulation methods, which construct Prolog procedures whose behavior models that of the specification, our simulator maintains execution traces of simulated protocol behavior and adds to these traces through the application of rules of inference reflecting the protocol behavior. This method provides a flexibility not found in other approaches.

1. Introduction

The problem of specification of the behavior of a system, without specifying the manner in which it is implemented, is a common one in computer science. Such problems are especially acute in those areas in which the primary point of the system being designed is communication with other systems, and where those other systems that communicate with the system in question must be designed on the basis of the specification alone, and must count upon the system conforming to the specified behavior. In many cases, the specified system is not yet built. Communications architectures and their associated protocols are one area in which this situation arises.

When developing a specification, the designer would often like to determine whether the system as specified will behave as desired. Later, the designers of other systems which will communicate with or otherwise use the specified system will find it desirable to test the interaction of their designs with the not-yet-built specified system. In either case, it is desirable to have a specification that is *executable*, that is, to automatically derive a system that behaves identically to the completed system (except for efficiency considerations affecting time and space usage). Some specification methods [5, 10], for whatever reason, do not provide executable specifications, while others [1, 3], provide executable, albeit inefficient, specifications.

Because specifications define behaviors, rather than implementations (in other words, because they specify *what happens*, rather than *how it happens*), they often have the flavor of declarative languages. This, in turn, leads to their being modeled through translation to declarative languages, in the case of the

examples discussed in this paper, to Prolog. A number of systems have been proposed that employ Prolog in the specification and simulation of communications architectures, either through directly modeling the specification in Prolog [12, 14, 18], or through translation from another specification method into Prolog [4, 13]. Other schemes, making use of the concurrent nature of communications architectures, employ concurrent logic programming languages such as PARLOG [11]. We feel that these schemes could in many cases just as easily and efficiently be implemented in some other languages, and do not take advantage of such distinctive Prolog features as unification, backtracking, and inferencing on a database.

The Cara simulator, part of the Cara system [9] developed at the IBM Zurich Research Laboratory, is an attempt to employ these features of Prolog to provide capabilities not available in other systems. The Cara simulator maintains a database containing the messages exchanged between simulated entities up to that point in the simulation, as well as information on previous entity states, and facts describing the configuration being simulated. The simulator then employs rules of behavior representing the specified communications behavior to infer new facts representing representing new state transitions and new messages exchanged by the simulated entities. The rules were written in a specially designed rule language, known as Carla [7], which is translated into Prolog. A general overview of the Cara system is given in [9]. The current paper discusses the organization of the simulator module in depth.

Simulating communications architectures through this inference-based model supports *exploratory simulation*, in which the user can easily experiment with various specification alternatives: the developer can stop the simulation, roll it back to a given point, modify the simulation, and resume. A designer can simulate starting from any intermediate state, even if the portion of the specification needed to reach that state has not yet been written. Similarly, the user can simulate a specification at a very high level, when formats of the data structures and messages, as well as large chunks of the specification, have not yet been defined. Finally, the use of Prolog allows the specifier to employ a logical, non-procedural style in the rule guards, which is easy to write and to understand.

The following sections describe the design considerations for our simulation method, as well as the method itself, and compare this method with other uses of Prolog for the specification of communications architectures.

2. Motivation

The simulation method we chose had to satisfy a number of requirements specified by the nature of the Cara system [9] of which it was a part. Cara is an environment for the graphical specification and simulation of communications architectures through the medium of message-flow diagrams [see figure 1]. It was intended that the users of the system be able to try out their specifications at an early stage in the design process, when only high-level specifications have been supplied, and when large segments of the specification are still unwritten. As we will see, the ability of Prolog to manipulate unspecified values through the use of unification and the logical variable is useful here, as is the lack of distinction between code and data; the latter allows us to replace entire Prolog routines temporarily by a single fact, or set of facts, functioning as a placeholder and simulating the behavior of the as-yet-unwritten routine at a high level.

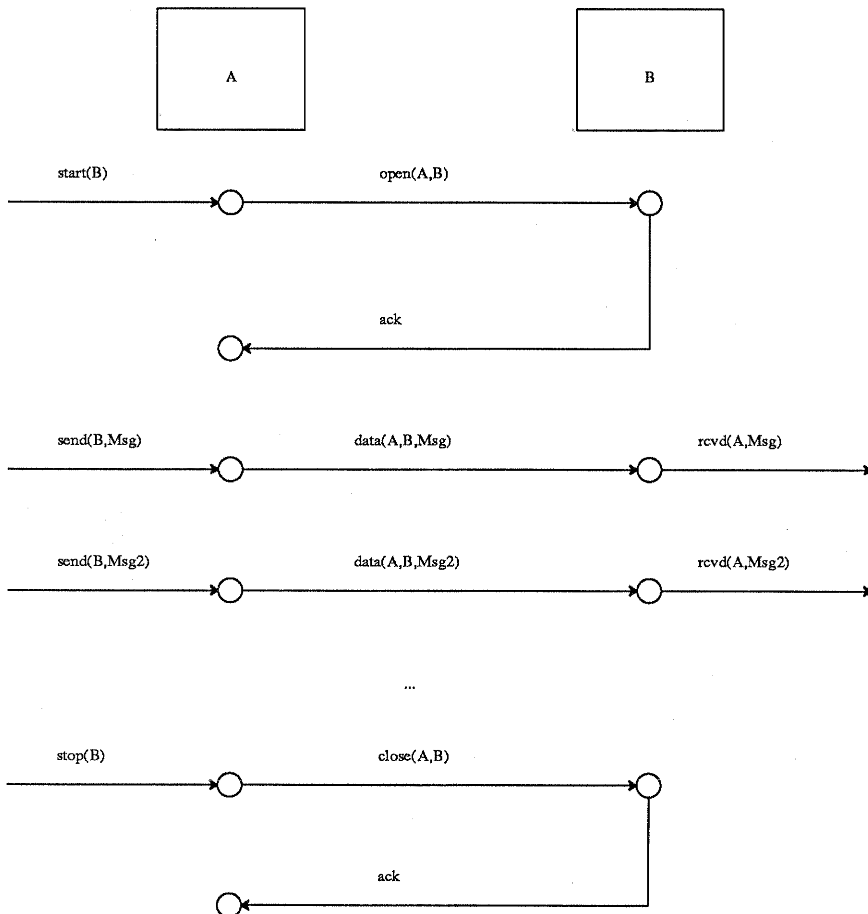


Figure 1 - a message-flow diagram

As mentioned earlier, one of the goals of Cara is to provide the user with an environment for exploratory simulation. This requires that the user be able to modify any part of the simulation at any time. In particular, since the specification representation is message-flow diagrams, the user must be able to edit any part of the diagram, including adding or deleting actions in the middle of the diagram. The use of a finite-state-machine-based approach would make such a facility difficult to implement, since changes to the specification could lead to extensive recalculation of states and transitions. A rule-based approach, on the other hand, would be appropriate, since there is no explicit or specific ordering between the rules, and adding or deleting actions in the diagram would result simply in adding or deleting the corresponding rules in the rule base.

Likewise, we planned that the Cara system should allow the user to specify complex sets of initial conditions (including entire histories of previous messages and receipts, and sequences of previous specification actions) on which the simulation could be performed. Since such ''canned histories'' might not correspond to any state in a finite-state machine corresponding to the possibly incomplete specification (and indeed, might never correspond to *any* state of the completed specification), an FSM-based approach would be inappropriate, while a rule-based approach, incorporating initial conditions and histories in the simulation database, was quite feasible.

Finally, Cara required that traces of a simulation be recorded for subsequent analysis. An FSM approach would require an external tracing mechanism, while the inference-based approach upon which we settled incorporated this as an inherent part of the simulation design, as will be shown later in the paper.

3. Inference-based simulation

3.1. Underlying model

The underlying model that we are simulating is communicating **protocol entities** [PEs] connected by **communications links** [commlinks, or simply links]. The PEs contain untyped **state variables** holding information partially describing the PEs' state. The remainder of the state consists of **history queues** recording the communications activity of the PE. PEs are connected to the links through **ports**.

Each PE belongs to a specific **PE type**, and all PEs of a given type exhibit the same structure (state variables and ports), and possess the same **rules of behavior**. These rules define the behavior of all PEs of that type.

Each rule consists of a **guard** and an **action**. Each PE possesses a **clock** of indeterminate rate. Each time the clock **ticks**, the guards of the rules are all evaluated. The evaluation may be done in any order or in parallel, since the guards have no side effects until a rule is **committed**. The guards may refer to state variables, message receipts, or previous message history. If no guard succeeds (that is, evaluates to **true**), no rule fires. Otherwise, one of the rules whose guard has succeeded is chosen, it is committed to, and its associated action is performed. This action can cause a message to be transmitted, or the PE's state to be altered. If more than one rule contains a guard that succeeds, this indicates an ambiguity or nondeterminism in the specification. In that case, the simulator can request user intervention, or can select one choice at random.

The commlinks are connections of indeterminate behavior and delay. Right now in our simulator, all commlinks are FIFO queues, with the delay controlled by the user, but we are experimenting with methods to allow us to specify other definite, or even nondeterminate, behaviors on the commlinks.

Figure 2 gives a diagram showing the structure of the execution model. It is described in more detail in [7].

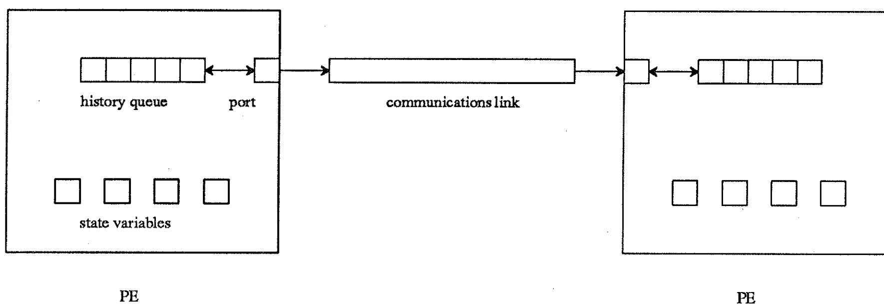


Figure 2 - Cara execution model

The organization and behavior of the model resembles that of Actors [2], although the original inspiration was that of infinitely recursive processes implemented using Guarded Horn Clauses [17] and

communicating through streams [16].

3.2. Simulation organization - databases

The databases maintained by the Cara simulator consist of two parts: an **architecture database** containing information on the structure and behavior of the PEs, and a **simulation database** containing information on the configuration currently being simulated, and on the course of that simulation over time.

In our implementation, employing Arity/Prolog, the databases are maintained as separate **worlds**. Note that there are many possible simulations of an architecture, and that by replacing the facts of the simulation database while keeping the facts and rules of the architecture database constant, the user can run multiple simulations of the same architecture.

Statements appearing in the architecture database describe PE types, their structure, and their properties. Four types of facts are found in the architecture database:

- **pe_type**(*PE_Type*). There is a PE type named *PE_Type*.
- **port**(*PE_Type,Port_Name,Number,IO*). Each PE of type *PE_Type* has *Number* ports named *Port_Name*. The type of the ports is *IO*, which may be either **input** or **output**. If *Number* is greater than 1, there is an array of **subscripted ports** numbered *Port_Name*.1 through *Port_Name.Number*. If *Number* is *, there is an unlimited number of ports named *Port_Name*, whose subscripts may be any atom.
- **state_var**(*PE_Type,Var_Name*). Each PE of type *PE_Type* has a state variable named *Var_Name*.
- **rule**(*PE_Type,ID,Rule*). Each PE of type *PE_Type* has a rule *Rule* with the unique identifier *ID*, which is used during the simulation to identify the rule. *Rule* is in Prolog, translated from the Carla language. The form of the rules will be discussed in section 3.3.

The statements appearing in the simulation database, unlike those in the architecture database, express the changes in the state of the simulation over time, and therefore have a time component. This component, called the **simulation time**, is an integer which expresses the number of ticks of a global clock since the beginning of the simulation. The first action of the simulation is considered to occur at time 1, although we shall see that initial conditions and ''prehistory'' can be assigned simulation time values less than 1. Every event expressed by a fact in the simulation database is assigned a simulation time value denoting when it occurs in relation to other events in the simulation. Simulation time is related to the graphical position of an event on the message-flow diagram drawn by the user. In addition to the simulation time, each PE keeps its own **local time**, which is a count of the number of evaluation cycles performed by the PE since its creation. The first evaluation cycle of the PE is assigned the value 1, although, again, prehistory can be constructed by assigning facts a local time less than 1.

Some simulation database facts also have a true/false value associated with them. Such a fact in the database denotes that the fact is known to be true or false, respectively, at the given simulation time. Declaring a fact to be false is different from removing it from the database (retracting it), which is the act of forgetting whether it was ever true or false.

Simulation statements are used to declare the existence of PE instances, describe their interconnections, and show the way in which the values of their state variables and ports change. (The changes in port

values represent message traffic.) The following types of facts may appear in the simulation database.

- **pe**(*PE_Type*,*PE_Name*,*Time*,*TF*). It is true/false (depending on the value of *TF*) that there exists a PE named *PE_Name* of type *PE_Type* at simulation time *Time*. This statement (and all other statements containing a *TF* value) is considered to be true until such time as a contradictory statement (same type and name, different *TF* value) becomes true.

- **commlink**(*Link_Name*,*Time*,*TF*). It is true/false that there is a communications link *Link_Name* at simulation time *Time*.

- **commlink_from**(*Link_Name*,*PE_Name*,Port_Name,*Time*,*TF*). It is true/false that a sending end of a communications link *Link_Name* is connected to the port Port_Name of PE *PE_Name* at simulation time *Time*. There may be multiple sending and receiving ends of a commlink, each connected to exactly one port. A message sent on any of the sending ends should be received at all the receiving ends (although there is nothing to guarantee that the message actually arrives at all - or even *any* - receiving end, or that it arrives at the same time at all ends on which it is received).

- **commlink_to**(*Link_Name*,*PE_Name*,*Port_Name*,*Time*,*TF*). It is true/false that the receiving end of a communications link *Link_Name* is connected to the port *Port_Name* of PE *PE_Name* at simulation time *Time*.

- **port**(*PE_name*,*Port_Name*,*Time*,*Value*,*Context*). The port *Port_Name* in the PE *PE_Name* has the value *Value* with context *Context* at simulation time *Time*. A context is an identifier attached to a message in order to associate it with other messages in a conversation. Unlike statements with a *TF* argument, the **port** statement is considered true only at *Time*. (In other words, values are not considered to persist until they are used or some other value arrives. If the value ought to be there at a later time, it is up to the user to place that fact in the database.)

- **state_var**(*PE_Name*,*Var_Name*,*Time*,*Value*). The state variable *Var_Name* belonging to the PE *PE_Name* has the value *Value* at simulation time *Time*. The statement is considered to be true for all subsequent times until a later statement concerning the variable becomes true. (Note that this statement, along with statements with a TF argument, are considered to have values that persist. This is simply a matter of convenience to avoid cluttering the database with many statements saying, for example, that a certain PE exists. Since, however, we cannot assume that commlinks behave this way, we cannot make that assumption for **port** statements. As we will see in the next section, the existence of persistent facts complicates somewhat the construction of rules of behavior.)

- **state_fact**(*PE_Name*,*Fact*,*Time*,*TF*). A fact, represented as a Prolog structure, is considered to be associated with the PE *PE_Name* (if *TF* is true) or not associated with the PE (if *TF* is false) at simulation time *Time*. *Fact* may be any value, completely or partially instantiated. Such statements are used to record information for later use by a PE, or to assert that certain "facts" are not true.

- **rule**(*PE_Name*,*RuleID*,*Time*). The PE *PE_Name* fired the rule with identifier *RuleID* at simulation time *Time*.

Statements may be explicitly asserted into the appropriate database, or implicitly asserted through inferences corresponding to firings of rules of behavior. Explicit assertions are performed through a simulator interface which will not be discussed here. The interface checks that the statements are syntactically

legal, that the arguments are validly used, and that a new statement is consistent with previously asserted statements. It is also to possible to query the databases to determine the truth or falsity of statements. These queries are also mediated by the interface to guarantee syntactic correctness, and also to account for the existence of persistent statements.

3.3. Rules of behavior

The rules of inference that describe the behavior of PEs are given in Prolog. However, since Prolog does not contain communications primitives or other facilities for describing communications architectures, we have designed a rule-based language called Carla. Carla rules are translated into Prolog and the Prolog rules are applied to the database. We will not describe Carla in detail here (it is described thoroughly in [7]), but will describe it sufficiently to understand the underlying Prolog structures.

A Carla rule consists of a **guard** and an **action**. The guard is an expression consisting of one or more terms connected by conjunctions and disjunctions. The guard terms include comparisons (of both **state variables** and **local variables**, that is, variables local to a particular rule), unifications, tests of whether a value has been received by the PE in the current evaluation cycle (i.e., whether it currently appears in the port), whether given state facts are currently true or false, and whether certain values have been received or sent by the PE sometime in the past. Guards have no side effects: they do not alter the values of state variables, or change the values in ports. When the simulator attempts to satisfy a term, the local variables in the term are unified with a possible solution to the term. If an attempt to satisfy a subsequent term fails, execution will backtrack to the most recent term for which another solution may exist, unbinding all the variable bindings made in the meantime. Thus, the evaluation of Carla guards uses the same mechanism as in Prolog. If a guard is satisfied (that is, if it evaluates to true), and the rule is **committed** (see below), then the particular solution found is also committed, and no further backtracking is allowed.

All rules for a given PE type are evaluated with respect to the given PE at the given time. Ideally, there will be exactly one rule tried whose guard succeeds. Since we allow, and in fact desire, the early simulation of incomplete specifications, it is possible that more or less than one rule have a successful guard. In the former case, the user will be asked by the simulator to choose one rule to fire, and the user has the opportunity to rewrite the rules so that in the future only one rule will succeed in this situation. The latter case indicates that either there is a missing rule, or the PE is in a state where it cannot do anything.

Once a rule is selected for firing, it is **committed**, and its **action** is performed. The action is a set of operations to be performed in parallel by the PE. (Actually, certain operations for calculating new values for local variables are done first, and sequentially, to avoid problems with data dependencies.) These operations include calculating new values for state variables, setting or unsetting state facts, and transmitting messages.

A few relevant features of Carla rules are:

- The $\rightarrow$ operator separates the guard from the action. The guard appears on the left of the $\rightarrow$, the action on the right.

- In the guard, the comma is used as the conjunction operator, and the semicolon is the disjunction operator.
- The := operator is used for assignment to a state variable, = is the unification operator, and == is arithmetic evaluation and comparison.
- **rcv**(*Port,Message,Context*) is true when a message *Message* with a context *Context* arrives in the port *Port* at the current time. Any uninstantiated arguments will be unified.
- **send**(*Port,Message,Context*) is an action that transmits a message *Message* with context *Context* through the port *Port*.
- **rcvd**(*Port,StartTime,Occur,Relative,Absolute,Message,Context*) is true if *Message* with *Context* was the *Occur*-th unifiable message (counting from the local time *StartTime*) where a negative value for *Occur* denotes a search towards the past, and a positive value denotes a search towards the future. A special *StartTime* of **now** refers to a search starting at the current local time. *Absolute* is the absolute local time or the reception (starting from time 0) and *Relative* is *Absolute - StartTime*. **rcvd** and **sent** (see below) are used for referring to message "history," that is, messages previously sent and received. Any uninstantiated variables will be unified, and backtracking will yield additional solutions if they exist. The complex nature of **rcvd** and **sent** allows them to be used in many different ways.
- **sent**(*Port,StartTime,Occur,Relative,Absolute,Message,Context*) is the counterpart of **rcvd** for transmitted messages.
- A **not** operator succeeds if and only if the expression which is its argument fails.

When a Carla rule is translated to Prolog, it takes the form of a pair

    ((Head :- Guard), Action)

It takes this form instead of the conventional

    Head :- Guard, Action

in order to allow guards to be separated from actions at those times when all relevant guards are executed. Conceptually, however, rules may be considered to take the latter form.

The functor of the rule head is the name of the PE type to which the rule applies. The head has two argument variables, PE_Name and CurTime, representing the PE for which the rule is attempting to fire, and the current simulator time. These variables are instantiated by the compiler at the time the rule is attempted. Thus, all rules for PEs for PEs of type p belong to procedure p/2.

Each Carla primitive has an equivalent Prolog structure to which the primitive is translated. These structures are connected with conjunctions and disjunctions in the same way as the corresponding Carla primitives. The primitives and their corresponding Carla structures are given in table 1. Note the use of the head arguments PE_Name and CurTime.

Note that **rcv** is simply a query to the simulation database that succeeds if a suitable port statement (representing the received message) is found. **send** is an assertion to that database (with some error checking).

Table 1a -
Carla guard primitives and equivalent Prolog structures

Carla primitives	Prolog structures
rcv(Port,Msg,Cxt)	port(PE_Name,Port,CurTime,Msg,Cxt)
rcvd(Port,Start,Occurs,Rel,Abs,Msg,Cxt) sent(Port,Start,Occurs,Rel,Abs,Msg,Cxt)	mhist(PE_Name,Port,Start,Occurs,Rel,Abs,CurTime,Msg,Cxt)
fact(X,Y,...) not(fact(X,Y,...))	state_fact(PE_Name,fact(X,Y,...),Time,TF), Time < CurTime, not(state_fact(PE_Name,fact(X,Y,...),T2,_), T2 > Time, CurTime > T2), TF = true /* or false, if **not** */
state variable reference (to state variable 'SV')	state_var(PE_Name,'SV',Value,Time), Time < CurTime, not(state_var(PE_Name,'SV',_,T2), T2 > Time, CurTime > T2)

Table 1b -
Carla action primitives and equivalent Prolog structures

Carla primitives	Prolog structures
send(Port,Msg,Cxt)	var(Port) -> <return error>; assert(port(PE_Name,Port,CurTime,Msg,Cxt))
fact(X,Y,...) not(fact(X,Y,...))	assert(state_fact(PE_Name,fact(X,Y,...),CurTime,true)), /* or false, if **not** */
state variable assignment 'SV' := <Value>	assert(state_var(PE_Name,'SV',Value,Time))

rcvd and **sent** are calls to the library routine mhist/9, which references the simulation database. Since transmitted and received messages differ only by whether they refer to an output or an input port, **sent** and **rcvd** call mhist/9 in exactly the same way. Likewise, since the issue of whether a transmission or receipt occurs in the past or the future depends only on the event time's relation to the CurTime value, past and future history (predictions that an event will occur, rather than a check that an event has occurred) are

referenced in the same way.

Assignments to state variables and assertions of state facts [table 1b] are simply assertions to the simulation database. Referencing state variables and facts is more complicated: code must be generated to locate the most recent relevant fact which is not later than the CurTime.

In addition to the arguments in the database structures shown, the structures contain transaction numbers assigned in the order in which the facts were asserted. This is useful in rolling back the database to a previous state. For reasons of clarity, we have left out the transaction numbers here.

3.4. Simulator operation

To simulate the behavior of a PE, the user, through a simulator interface command, requests the identifiers of all rules for the given PE whose guards are satisfied at the given simulation time. The simulator looks up the PE's type (the information is stored in the simulation database), calls all the clauses (corresponding to rules of behavior) in the procedure whose name is the PE type (using a **setof** predicate in order to find all solutions), and returns the identifiers of all rules whose guard succeeds. Rule guards attempt to satisfy themselves by directly querying the simulation database, searching for the appropriate **port**, **state_var**, and **state_fact** statements.

After the system notifies the user of those rules that can fire, the user chooses one of the rules. The simulator then executes the rule's action, which updates the simulation database, recording the message transmissions and state changes performed by the rule.

The user can also simulate the behavior of a rule that has not yet been defined by directly asserting **port**, **state_var**, **state_fact**, and even **rule** statements into the simulation database. Simulator interface commands exist to accomplish this, as well as to retract such statements from the database.

The user repeatedly attempts to fire different PEs in the simulated configuration, using the graphic interface described in [9]. The system front end translates user actions into the appropriate simulator commands. There is a link simulator that is responsible for simulating the transport of messages across links, or the user can simulate this by hand, by graphically demonstrating the movement of the message. In either case, the system queries the simulation database to find the **port** statement representing the sent message, and asserts a new **port** statement specifying the receiving PE, along with the port and time of receipt. The message can now influence the firing of a rule in the receiving PE.

3.5. Simulator capabilities

The simulator scheme presented here fulfills the requirements laid out in section 2.0. The paradigm of "exploratory simulation" is supported in a number of ways. For example, it is a simple matter for a user to determine what would happen in the simulation if certain previous actions had never occurred, or had occurred differently. This is done by retracting those statements representing the previous actions (for example, messages received or sent), and asserting new statements representing the new "previous actions." Likewise, an entire "canned simulation" can be asserted into the database, and the simulation taken up from there. This can be useful in simulating a part of a protocol when the part representing the earlier history has not been formally specified, or for simulating from a state that may not be reachable

from normal initial conditions. It can also be used to save a simulation and resume further simulation at a later time. None of the simulation schemes mentioned in section 5 allow such actions.

It should be noted that the facts in the simulation database represent a complete trace of the simulation so far. This can be printed out (either textually or graphically), analyzed, stored in a file, and reloaded into the simulation database for further simulation.

The Cara simulator allows multiple designs to be simulated on the same data and the results compared. For a given set of initial conditions, one design can be loaded into the architecture database and simulated. Then, the initial conditions can be re-established in the simulation database and a new design read into the architecture database and simulated. This process can be repeated for each design. It is also possible to easily and quickly change a design by retracting and asserting rules of behavior, or PE type, state variable, and port declarations in the architecture database, even while the simulation is under way.

High-level specifications can be simulated in the Cara simulator, even if the underlying low-level specification has not yet been written. For example, a design may require a routine that computes all the other PEs to which the given PE is connected. If that code has not yet been written, a set of state facts giving the connectivity for that simulation can be substituted for it and asserted into the simulation database. Since state fact checks are identical in form to procedure calls (both are **call** subgoals in Prolog), low-level routines can later be substituted for the state facts without altering the rules.

Finally, as mentioned before, incomplete specifications can be simulated. If a rule does not yet exist, its effect can be simulated by the user through the direct assertion of those facts that the rule would have asserted had it existed. These direct assertions can be effected through the graphic user interface, or through the textual simulator interface commands.

4. An example

We will illustrate the use of the simulator by simulating the alternating bit protocol [15], a simple, not very efficient protocol for transmitting data across noisy channels.

In the alternating bit protocol, the sender and receiver each possess a bit (called S and R, respectively). The S bit is the tag to be attached to the next message to be sent; the R bit is the tag of the next message expected to be received. (Entities may possess both S and R bits, and consequently play the role of both sender and receiver.)

The main idea of the protocol is simple: an entity may only transmit a message if an acknowledgement for the previous message has been received, and an entity may only receive and acknowledge a message whose tag is equal to the tag value it expects (the R bit). Upon transmitting or receiving messages, the S or R bits, respectively, are flipped. S and R are both initially 0.

Six rules of behavior describe the protocol. (N.B.: there are some errors deliberately introduced in these rules. We will use the simulation to discover them.) Informally in English, they are:

1) Transmit a message: If there is a message to be sent, and either the previous message sent has been acknowledged or this is the very first message, transmit the message along with the S bit, and flip the S bit.

2) Receive a message: If a message is received whose tag is equal to the R bit, relay it to the user, send an acknowledgement containing the received tag, and flip the R bit.

3) Retransmit a message: If no acknowledgement has been received for the previous message sent, resend the previous message along with the previous S bit (the inverse of the current S bit).

4) Retransmit an acknowledgement: Send an acknowledgement of the previous message (i.e., containing the inverse of the current R bit).

5) Discard a message: If a message is received whose tag is not equal to the R bit, do not relay it, and do not acknowledge it.

6) Receive an acknowledgement: If an acknowledgement of the previous message is received, record it.

We first describe the structure of a PE by asserting facts about its type to the architecture database:

```
pe_type(pe).
state_var(pe,'S').
state_var(pe,'R').
port(pe,from_user,1,in). /* messages from the user */
port(pe,to_user,1,out).         /* messages to the user */
port(pe,from_pe,1,in).          /* messages from the other pe */
port(pe,to_pe,1,out).           /* messages to the other pe */
```

We next specify the rules of behavior in Carla:

```
1) rcv(from_user,M,C),
   ((sent(to_pe,now,-1,_,_,message(_,Last_S),C),
    rcvd(from_pe,now,-1,_,_,ack(Last_S),C)); init)
   → send(to_pe,message(M,S),C), S := ^S.
```

Note: new values of state variables (e.g., S in the rule above) take effect only after the other actions have been completed.

```
2) rcv(from_pe,message(M,R),C)
   → send(to_user,M,C), send(to_pe,ack(R),C), R := ^R.

3) sent(to_pe,now,-1,_,_,message(M,Last_S),C), Last_S is ^S,
  not(rcvd(from_pe,now,-1,_,_,ack(Last_S),C))
  → send(to_pe,message(M,last_S),C).

4) true → send(to_pe,ack(Last_R),C) where last_R is ^R.

5) rcv(from_pe,message(M,Tag),C), not(Tag = R) → true.

6) rcv(to_pe,ack(Tag),C) → true.
```

The Prolog translation for the first rule is:

```
Head: pe(PE_Name,CurTime) :-
        port(PE_Name,from_User,CurTime,M,C),
         /* test of sent and rcvd */
         ((mhist(PE_Name,to_pe,CurTime,-1,_,_,CurTime,message(_,Last_S),C)
          mhist(PE_Name,from_pe,CurTime,-1,_,_,CurTime,ack(Last_S),C));
          /* test of init fact */
          (state_fact(PE_Name,init,Time,TF),
           Time < CurTime,
           not(
             state_fact(PE_Name,init,T2,_),
             T2 > Time, CurTime > T2),
          TF = true)).

Action: /* compute current and new values of S */
         state_var(PE_Name,'S',S,Time2),
         Time2 < CurTime,
         not(
           state_var(PE_Name,'S',_,T3),
           T3 > Time2, CurTime > T3),
         NewS is 1-S,

         /* send message.  assert new value of S */
         assert(port(PE_Name,to_pe,CurTime,message(M,S),C)),
         assert(state_var(PE_Name,'S',NewS,CurTime)).
```

The other Prolog rules may be similarly constructed.

We now set up a test configuration by asserting facts into the simulation database. We must define the two PEs and their connection:

```
pe(pe,pe1,0,true).
pe(pe,pe2,0,true).
commlink(connect1).
commlink(connect2).
commlink_from(connect1,pe1,to_pe,0,true).
commlink_to(connect1,pe2,from_pe,0,true).
commlink_from(connect2,pe2,to_pe,0,true).
commlink_to(connect2,pe1,from_pe,0,true).
```

We choose to test the protocol by setting up initial conditions. These include initial values for S and R, the init indicator, and an initial message:

```
state_var(pe1,'S',-1,0).
state_var(pe1,'R',-1,0).
state_var(pe2,'S',-1,0).
state_var(pe2,'R',-1,0).
state_fact(pe1,init,-1,true).
port(pe1,from_user,0,first_msg,c1).
```

The logical first step is to learn what may happen at time 0, so we issue a command to the simulator requesting this information. The simulator translates it into a Prolog query that attempts to satisfy all the guards with CurTime instantiated to 0. The simulator tells us that pe1 may fire rule 1, and that both pe1 and pe2 may fire rule 4. We have found our first error: PEs should not acknowledge non-existent messages. We retract rule 4 and replace it with

```
4) rcvd(from_pe,now,-1,_,_,message(_,Last_R),C)
  → send(to_pe,ack(Last_R),C).
```

Now a PE will only acknowledge a message it has received.

We now fire rule 1 on pe1, which causes a message to be sent, and the value of S to be changed:

```
port(pe1,to_pe,0,message(first_msg,0),c1).
state_var(pe1,'S',0,1).
state_fact(pe1,init,0,false).
```

In the absence of a link simulator, the user transports the message from pe1 to pe2. Generally this may be done graphically, by drawing a line, but this is translated to a simulator command that asserts the fact

```
port(pe2,from_pe,1,message(first_msg,0),c1).
```

We ask the simulator what may happen at time 1 and the simulator replies that pe2 may fire rule 2 (it receives the message), and that pe1 may fire rule 3 (it attempts to retransmit the message). We note that it is a bit premature for pe1 to retransmit the message and make a note to revise rule 3 to include a timeout. For the moment, we choose to assume that pe1 performs correctly and we therefore ignore its rule firing and merely fire pe2's rule 2. Further simulation continues similarly.

We can set up intermediate states in order to test conjectures concerning protocol behavior. Perhaps we suspect that rule 4 is still incorrect. We empty the simulation database and start over. We create a single PE p. We assume it has received and acknowledged a message with tag 0 and now receives another message with tag 0. Will it handle it correctly? We assert the following facts to start:

```
pe(pe,p,-1,true).
port(p,from_pe,0,message(m1,0),c).
port(p,to_pe,0,ack(0),c).
port(p,from_pe,1,message(_,0),c).
```

Note that we don't care how we reached this point (hence the sender PE is not shown), nor do is it relevant to show the message being relayed to the user. (The organization of the simulator makes it unnecessary to specify the entire configuration.) Likewise, we don't care what the message received at time 1 is, so we leave it blank. We also don't bother to assign a value to the variable 'S'. We ask the simulator what rule(s) p may fire at time 1. The answer is that rule 5 may fire (the received message is discarded), and so may rule 4. Although the two rules do not interfere with each other, our model allows only one rule at a time to fire, so we must make a note to reformulate the rules to allow 5 to fire in this situation, but not 4. (Perhaps 4's guard should require that no message be waiting to be received.) We explicitly direct that 5 be fired, then ask the simulator what may happen at time 2. The answer is that only 4 may fire. The rule behaves as expected.

5. Comparison with other work

Two of the major types of formalisms for the specification of communications architectures are finite-state machines [FSMs], typified by Estelle [1], and process algebra, typified by LOTOS [3]. Simulation systems for both have been implemented in Prolog. We will look at one example of each, although there are a number of others [4, 6, 12, 14].

Von Bochmann *et al* [18] proposed the modeling of FSM state transition rules in Prolog. Each transition rule was represented by a Prolog rule of the form

```
t(Present_State,Next_State,Input,Output) :-
  enabling_condition,
  action.
```

Input and output queues are represented as lists. A network of communicating FSMs is represented by a rule presenting the FSMs connected by logical variables. For example, in the network shown in figure 3, the transition rules for the whole system are written as follows (note that the state of the system consists of the states of the two components plus the contents of the queues):

```
t_S([PS_A,PS_B,Q_A_to_B,Q_B_to_A],
   [NS_A,NS_B,NQ_A_to_B,NQ_B_to_A],
   [To_A,To_B],[From_A,From_B]) :-
     Q_B_to_A = [B_to_A|IQ_B_to_A],
     Q_A_to_B = [A_to_B|IQ_A_to_B],
     t_A([PS_A,NS_A,[To_A,B_to_A],[From_A,B_from_A]),
     t_B([PS_B,NS_B,[To_B,A_to_B],[From_B,A_from_B]),
     append(IQ_B_to_A,[A_from_B],NQ_B_to_A),
     append(IQ_A_to_B,[B_from_A],NQ_A_to_B).
```

Prolog-based specifications of this type are suitable for testing whether a sequence of inputs is legal under the specified protocol, and 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 Cara approach, it is unsuitable for exploratory development and simulation of communications architectures. It is not possible to simulate incomplete protocols,

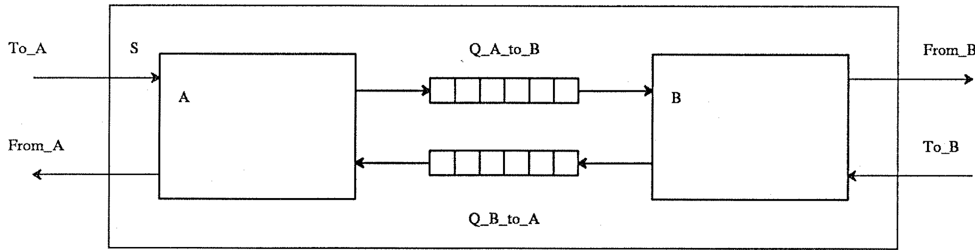


Figure 3 - Sample simulated network

nor is it possible to modify the transition rules during the simulation. It would be very difficult to alter simulation state during the simulation, or to initialize the simulation to a particular state. And since the entire system state is incorporated in the machine state and the contents of the system queues, it is impossible to design specifications that refer explicitly to previous input and output history, thus ruling out a very convenient specification style.

Pappalardo [13] proposed a Prolog-based implementation of ECCS, a process algebra-based specification language related to LOTOS. Using ECCS or LOTOS, a designer can specify the acceptable communications behavior between a number of entities. Given the ECCS specification, a simulator can be written in Prolog to reflect its nondeterministic behavior. This simulator is written in the form of derivation rules:

```
der(P,A,Q)
```

in which some process expression P performs action A and becomes process expression Q. For example, the sequencing operator A;P is expressed in Prolog as

```
der((A;P),A,P)
```

or, A;P performs A and becomes P. The meaning of the alternative operator P1+P2 is expressed as

```
der(P1+P2, A, Q) :- der(P1, A, Q).
der(P1+P2, A, Q) :- der(P2, A, Q).
```

Note that the above expression is nondeterministic. There are more complex constructs expressing parallel execution, or communication between two entities. The above system allows nondeterminism and maintains the necessary choice points, something that, in the Cara simulator, must be taken care of by the user. In the Cara system, when there are two possible choices, the system so informs the user, but the user must keep track of these points and explicitly roll the simulation back to that point (not difficult in itself) in order to try an alternative.

In Pappalardo's system, since the specification is bound to logical variables in the derivation rules, it is not possible to alter the specification or any execution information while the simulation is under way. It is also not possible to reference history. Finally, it is necessary to instrument the derivation rules if we wish to produce a trace of the executing protocol.

6. Current status and future work

The Cara simulator has been implemented in Arity/Prolog and runs on an IBM PS/2 running OS/2. (We are currently porting it to Quintus Prolog running on a Sun SPARCstation.) It is part of the Cara system [9], a graphical development environment for communications architectures based on message-flow diagrams. The diagrams used in Cara, however, are not the specification itself; the system extensively interacts with the user to derive an executable specification from the diagrams. We are currently developing a visual language based on message-flow diagrams that completely and unambiguously specifies the communications architecture being designed [8]. We expect the Cara simulator to be part of the development environment for this language.

The simulator itself will have to be improved in a number of ways. First, link behavior will have to be modeled automatically. We have designed a rudimentary link simulator that simulates a wide class of link behavior, but it does not incorporate the randomness that we would like to employ in modeling certain classes of links. For completely deterministic links, such as ideal FIFO queues that never lose or corrupt a message and have known timing characteristics, the link simulator is satisfactory. Modeling such links may be useful during the protocol's design phase, but during testing we would like to test the robustness of protocols under various unanticipated conditions.

The second way in which the Cara simulator must be improved is in the management of structure in the specification. The simulator currently handles flat specifications. Most communications architectures contain many levels, sometimes seven or more, arranged in a ''protocol stack.'' It is currently possible for us to develop and simulate each of these levels separately, but we are unable to simulate the interaction between the various levels, or to show that the multiple levels are consistent in their behavior.

7. Concluding remarks

We have outlined the design of a simulator for communications architectures written in Prolog. Through the use of Prolog's ability to manipulate facts in a database and to compute inferences based on those facts, our simulator supports ''exploratory simulation'' of protocols at a very early stage in their development: incomplete and high-level designs may be simulated; both the design and the data on which it operates may be changed at any time, even during the simulation; multiple designs may be run on the same initial data; traces of the behavior may be generated; and a style of specification involving reference to previous, or even future, communication history is possible. Such flexibility is not present in other Prolog-based specification simulation systems, because they attempt to model the specification directly as a Prolog equivalent, and because the specification data is bound to the logical variables of the specification during the simulation. This flexibility allows us to make full use of the special features of Prolog in a way the other methods do not, and will contribute heavily to the visual protocol specification language currently under development.

Acknowledgements

The work described in this paper was performed by the author during his term as a post-doctoral researcher at the IBM Zurich Research Laboratory. The author benefited from extensive conversations with colleagues in the Cara project and in the Communications Architecture Software and Technology

group. Special thanks go to Alistair Cockburn, who collaborated with me on the design of the Carla language, and who made many important contributions during the design of the simulator interface, and to Liba Svobodova, who made many useful comments on an earlier draft of this paper.

References

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

2. G. Agha, *Actors: a model of concurrent computation in distributed systems,* University of Michigan, Ann Arbor, MI, 1985. PhD thesis

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

4. J.P. Briand, M.C. Fehri, L. Logrippo, and A. Obaid, ''Executing LOTOS Specifications,'' *Protocol Specification, Testing, and Verification, VI*, pp. 73-84, Elsevier Science Publishers B.V. (North-Holland), 1987 .

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

6. N. Choquet, L. Fribourg, and A. Mauboussin, ''Runnable Protocol Specifications Using the Logic Interpreter SLOG,'' *Protocol Specification, Testing, and Verification, V*, pp. 149-168, Elsevier Science Publishers B.V. (North-Holland), 1987.

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

8. W. Citrin, ''Design Considerations for a Visual Language for Communications Architecture Specifications,'' *Proceedings 1991 IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991. To appear

9. A.A.R. Cockburn, W. Citrin, R.F.Hauser, and J. von Kaenel, ''An Environment for Interactive Design of Communications Architectures,'' *Proc. 10th Intl. Symposium on Protocol Specification, Testing, and Verification*, Ottawa, June 1990.

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

11. D. Gilbert, ''Executable LOTOS: Using Parlog to Implement an FDT,'' *Protocol Specification, Testing, and Verification, VII*, pp. 77-88, Elsevier Science Publishers B.V. (North-Holland), 1987.

12. L. Logrippo, D. Simon, and H. Ural, ''Executable Description of the OSI Transport Service in Prolog,'' *Protocol Specification, Testing, and Verification, IV*, pp. 279-293, Elsevier Science Publishers B.V. (North-Holland), 1985 .

13. G. Pappalardo, ''Experiences with a Verification and Simulation Tool for Behavioral Languages,'' *Protocol Specification, Testing, and Verification, VII*, pp. 77-88, Elsevier Science Publishers B.V. (North-Holland), 1987.

14. D. P. Sidhu, ''Protocol Verification via Executable Logic Specifications,'' *Protocol Specification, Testing, and Verification, III*, pp. 237-248, Elsevier Science Publishers B.V. (North-Holland), 1983 .

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

16. E. D. Tribble, M. S. Miller, K. Kahn, D. G. Bobrow, and C. Abbott, ''Channels: A Generalization of Streams,'' *Concurrent Prolog: Collected Papers*, vol. 1, pp. 446-463, MIT Press, Cambridge, MA, 1987.

17. K. Ueda, ''Guarded Horn Clauses,'' *Concurrent Prolog: Collected Papers*, vol. 1, pp. 140-156, MIT Press, Cambridge, MA, 1987.

18. G. von Bochmann, R. Dssouli, W. Lopes de Souza, B. Sarikaya, and H. Ural, ''Use of Prolog for Building Protocol Design Tools,'' *Protocol Specification, Testing, and Verification, V*, pp. 131-147, Elsevier Science Publishers B.V. (North-Holland), 1986 .