January 1987

Hcport No.STAN-CS-87-1148
Also numbered KSL-86-36

An Instrumented Architectural Simulation System

by

Bruce A. **Delagi**, Nakul **Saraiya**, Sayuri Nishimura, and **Greg** Byrd

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL 86-36

January 1987

An Instrumented Architectural Simulation System

by

Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

WORKSYSTEMS ENGINEERING GROUP
Low End Systems and Technology
Digital Equipment Corporation
Maynard, Massachusetts 01754

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University Department of Electrical Engineering.

Table of Contents

List of Figures

ABSTRACT

AN INSTRUMENTED ARCHITECTURAL SIMULATION SYSTEM

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on the problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement. A simulation system with these goals is described together with the approach to its implementation. Its application to the study of a particular class of multiprocessor hardware system architectures is illustrated.

1 INTRODUCTION

Simulation systems are quite often developed in the context of a particular problem. **To** a degree, this is true for SIMPLE, an event based simulation system, and CARE, the computer array emulator that runs on SIMPLE.[1] The problem motivating the development of both SIMPLE and CARE was the performance study of 100 to 1000-element multiprocessor systems executing a set of signal interpretation applications implemented as "1000 rule equivalent expert systems" [2 3.

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represented significant bodies of code and so simulation run times were expected to be an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements were sufficiently unexplored prior to simulation that simplifications in the CARE system model, specifically with respect to element interactions, were suspect. This need for detail was, of course, in tension with the need for simulation performance. The ways that simulated system components would be composed into complete systems was initially difficult to bound. Further, it was clear that the models of these components would be elaborated over time and would undergo substantial change as design concepts evolved. It was also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicated what alternative aspect of system operation *should* have been monitored in any given completed run.

The design goals that emerged then were (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

1.1 Design Time Interaction And Run Time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage design evolution. Such encapsulation partitions the design along well defi ned boundaries. Components (by and large) interact with other components only through defined *ports.* Connections between components terminate at such ports. When a system simulation is initializecl, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of **system structure, component behavior**, and **instrumentation** into separate domains of consideration helps in managing a design that is both fluid and complex. **System structure,** that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. **Component behavior** is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE simulated system is a member of a class defined for that component type. **Instrumentation** is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general *instrumented-box* class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made a separate, substantially independent consideration in the design.

---

[1] SIMPLE and CARE were developed by the authors at the Knowledge Systems Lab of Stanford University. SIMPLE is a descendent of PALLADIO [1] optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. It is written in Zetalisp [4] and currently runs on Symbolics 3600 machines and TI Explorers.

A further partitioning of concerns is employed to separate out the definition of the application programming language interface and its support (as provided by CARE) from the underlying information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The *component probe* definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turned out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular *instrument panels* and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the *instrument specification.* This is a definition of what kinds of panels are included in an *instrument, how* they fit on an *instrument screen,* how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel, The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.

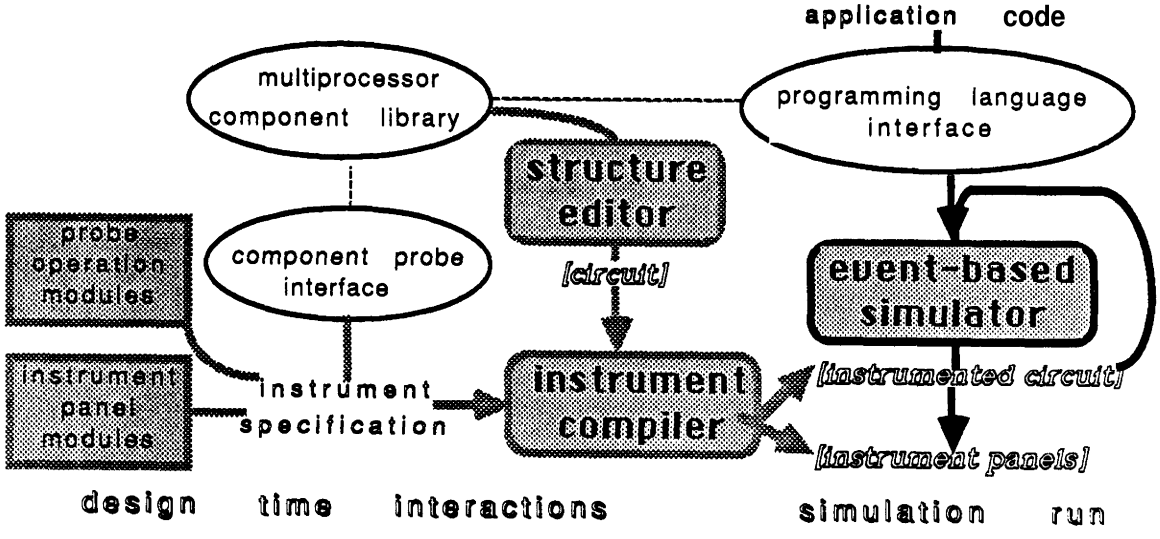

Figure I: Design Time Interactions and Run Time Representations

Putting together all the definitions of components, component probes, panels, instruments. applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 1 illustrates the partition between design time interactions and simulation run time operation. Structure editing pulls together components from the component library to produce a *circuit.* Associated with some components in the library, there are definitions for the syntax and underlying mechanisms of a multiprocessor applications language. These specify the

interface used to provide the program input to the multiprocessor system being simulated.[2] The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of probe *operation modules* to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an *instrumented circuit* and an *instrument*. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument **screen.** The instrumented circuit ties together instances of components, probes, and panels for a simulation run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

2 STRUCTURE AND COMPOSITION

Design time interactions to specify a system include the establishment of component relationships. Such specifications can be said to accomplish the composition of the system from its components and so define its structure. SIMPLE supports hierarchical composition: components may be described in terms of a fixed set of relationships among their sub-components. Additionally, such composite components may have function beyond what can be inferred strictly from their composition. All this can then be included a higher level composite (as shown in figure 2) and so on indefinitely until the top level "circuit", the system structure, is reached.

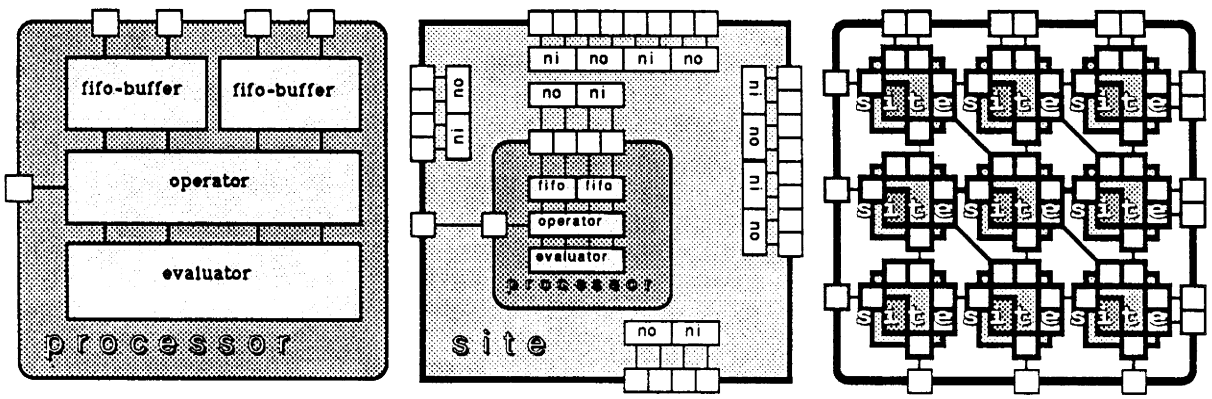

Figure 2: Hierarchical Composition

The behavior induced on a composite component from its parts changes according to the behavior of its parts. Thus, for example in figure 2, if at any time during a simulation the function of CARE *operator* components is changed by redefining their operation, the behavior

[2]The language primitives supplied can be used to define multiprocessor language interfaces for either shared-variable or value-passing paradigms. As supplied, the language interface built on these primitives supports value-passing on streams between objects but alternative interfaces can be (and have been) easily defined in terms of the given primilives.

of the nine-site grid is in immediate correspondence.[3]

Composition is described graphically and interactively in SIMPLE by picking a previously specified component type from a menu, placing it in relationship to other components with "mouse" movements, and, through the same means, specifying the connections between its selected ports and those of other components (as indicated in figure 3).

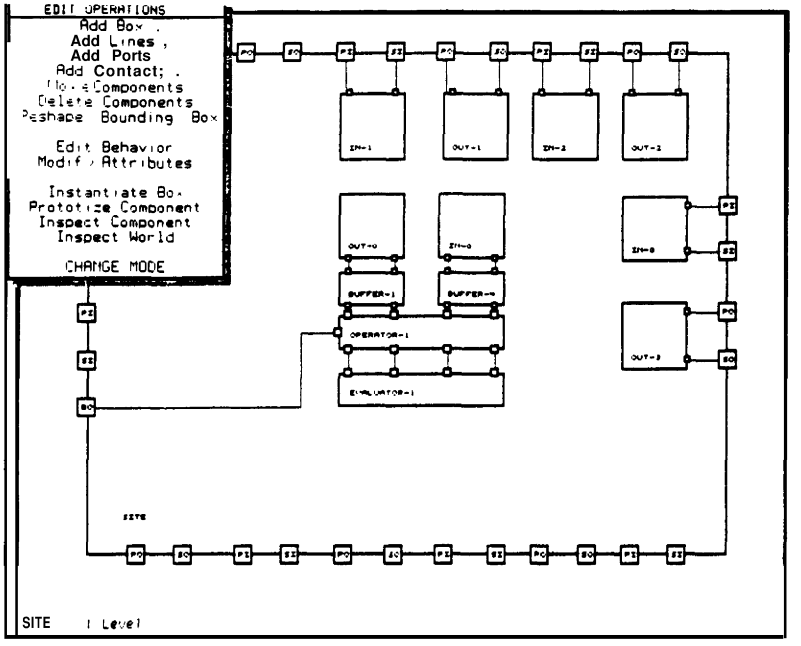

Figure 3: Graphic Structure Specification

Through another menu selection, ports can be defined for the new composite component so that it, in turn, can be fitted into yet higher level structures. Such external ports can be connected directly to ports of sub-components "within" the composite. If this is done, information appearing on that external port will be the responsibility of the connected sub-component. By this same means, a component previously described as a base level component, can be redefined as a composite of yet lower level elements as its design is elaborated with further details.

Components and (internal) connections can also be deleted from a library component and replaced with substitute components. After all sub-components and connections have been added, deleted, elaborated, and replaced as required, the completed structure can then be entered into a library of components and used in turn to compose higher or equivalent level components.

2.1 CARE Base Components

CARE supplies a small library of system level base component types. Currently these are the *net-input,* the *net-output,* the *fifo-buffer*, the *operator*, and the *evaluator*. The net-input, net-

[3] However, for reasons concerning simulation performance and because of their relatively low frequency, changes in the number and names of the internal state variables of components and the structural relationships between sub-components of a composite are not reflected in an already instantiated circuit. Changes in the internal structure of a CARE *site* library component, for example, will be reflected only in circuits instantiated after the change took effect. For this reason and to reduce long term storage requirements and load time for the fundamentally iterative circuits that we primarily study, we do not keep files of instantiated circuits. They are instantiated as needed from a high level library component with the same prototypical structure.

output and fifo-buffer accept (or block), route, and buffer transmissions. They do so in accordance with a dynamic, flow-controlled, multicast, cut-through communications protocol as described in [3]. The evaluator does the real work of the application: evaluating the application of functions to their parameters. The operator does the overhead work associated with such evaluations: for example, scheduling processes and sending and receiving (but not routing) messages.

In keeping with the objective of focusing simulation cycles on the aspects of the simulation particularly relevant to multiprocessor operation, the behaviors of the net-input, net-output, and fifo-buffer component classes are defined in fair detail, that is, at the register transfer level. Routing operations are described procedurally and assumed to occur within a time set by a parameter to the simulation. As indicated previously, the simulation of the operator and evaluator is broken into two aspects: the control of the flow of information and the functions performed on that information. The former is described in terms of SIMPLE behavior rules (as documented in section 3), register transfer by register transfer. The latter is described directly in terms of procedures and the simulated time taken by such procedures is modeled. In the case of the operator, this is done as a function of the number of storage cells manipulated during an operator procedure. In the case of the evaluator, this is done as a function of the execution time used by the machine executing the simulation, that is, the simulation vehicle.

2.2 CARE Composite Components

The prototypical composite component supplied with CARE is the *site*. As supplied, it includes net-inputs and net-outputs for up to eight "neighboring" components (generally other sites), a net-input and a net-output with associated fife-buffers for local receptions and transmissions, and, finally, an operator and evaluator as described above. Specializations of the site, for example, the *torus-situ*, exist in the library to fit the site into alternative topologies by supplementing the site routing and wiring procedures as appropriate to the topology.

2.3 Automatic Composition in CARE

Although any connection of components can be created by the means noted previously, for some repetitive, well patterned systems of connections, composition can be automated. The CARE library includes a component, the *iterated-cell*, which represents a template for the creation of composite components by iteration of a unit cell. The unit ceils (for example, the torus-site) are specializations of other components (for example, the site) as just discussed. The specializations include a method for responding to a request to provide a wiring list. Such a list associates each source port of a cell with the corresponding destination port (in terms of port names) and the position of the destination cell relative to the source cell in the iterated structure. The iterated cell component uses this information to make the required connections between each of its constituent cells.

3 SPECIFYING BEHAVIOR

SIMPLE is an event based simulator. The behavior of a simulated component is described in terms of responses to the events pertinent to that component. A component's response may include consequent events to be handled by the simulator as well as direct operations on component state. Assertion of consequent events and the responses to them (involving further consequences) drives the simulation. When there are no more events to handle, the simulation is complete.

To maintain modularity in a simulation system, responses to simulation events should be local to the affected component and its defined ports, that is, its connection to the remainder of the simulated system. The composition system of the simulator maintains the relationship between ports of one component and those of other components connected to them. Assertions

relative to a port of a component are thus systematically translated to events pertinent to components connected to it. This is the general mechanism for event propagation between components. In a limited number of cases, a direct operation on a related component may be appropriate. With fair warning about its possibility of abuse, a facility is provided to accomplish this.

3. I Behavioral Rules

The behavior of a component is described in terms of its responses to pertinent events. Each event stipulates the component affected, its port or state variable signalled with an assertion, the asserted value, and the simulated "time" of the event. The time of an event may be thought of as the "current" simulation time. Differences in event times represent the temporal relationship between events. Event times in SIMPLE simulations are monotonically increasing.

For each type of component, there is a procedure to handle pertinent events. The arguments to the procedure are those stipulated by the event (as just described). The procedure tests for conditions and, as satisfied, asserts or directly effects consequent actions. The conditions may include arbitrary predicates on the event parameters and the state variables of the component.

Event based simulators are based on the assumption that state and port variables remain unchanged until explicitly modified. Synchronous designs, that is, those in which the opportunities for state change are temporally quantized to a clock, can be modeled in such implicitly asynchronous, event based simulators by asserting the clock signal on a port of each and every clocked component of the simulated system. If only some of the components in a system need take action on each clock signal, there is an obvious inefficiency in this approach that is crippling for systems with even a modest number of components.

If, however, event times in an event based simulator are restricted to integers, the clock can be assumed. All that is needed is a way to detect the event for which a boolean combination of conditions as strobed by an assumed clock is first met. Primitive condition predicates are supplied for detecting an "edge" (a value changed by the current event) with a coincident "level" (a value set before the current even&) of two ports or state variables of a component in either of the two possible event sequences. The predicate both-states in the example evaluator behavior rule shown in figure 4 has these semantics.

```
;; If the evaluator is ready and there is at least one runnable process...
((or (both-states Evaluator-Status[4] 'ready Evaluator-Queue-Status 'some)
     (both-states Evaluator-Status 'ready Evaluator-Queue-Status 'full))
;;; make it current, start evaluation, and adjust status as per removal.
 (setq Evaluator-Status 'busy)                              ; block rule
 (assert-state Evaluator-Status 'busy now)                  ; next event
 (setq Current-Evaluation (queue-take Evaluator-Queue))     ; note process
 (user-evaluate Current-Evaluation now)                     ; execute it
 (send self :evaluator-queue-decreased now))                ; note change
```

Figure 4: Example Condition/Action Behavior Rule

Figure 4 illustrates the generality of SIMPLE behavioral descriptions. The underlying object-oriented programming system, Flavors [4], in which SIMPLE is implemented provides for direct reference of component state variables. The conditions and actions of behavior rules for a component then need only name the component's port or state variable (as stipulated in the definition of that component type) to get or change the appropriate value in the component instance for which the event is pertinent. Actions may include arbitrary procedures: for example, the procedures user-eval uate and queue- take in the given example.

[4] By convention, component state variables are written in capitalized form.

3.2 Using Methods

The environment for the execution of the procedures defining responses to events includes the state variables and ports of the component instance for which the event is pertinent. These procedures are Flavor *methods* [4] (in this case corresponding to the :ApplyRules message) of the component type and, as just noted, refer implicitly to the state variables of the component instance handling the event. Other methods may be defined for simulated components: for example, the : evaluator-queue-decreased method invoked in figure 4. Such methods have proved to be a natural way to realize the functional operations of components not described by behavior rules.

The composition system leaves information about the enclosing and contained component instances for each simulated component in system defined state variables of that component. With this information, methods directly referencing the ports and state variables of such related components may be invoked as needed. This is a useful but sharp-edged facility. The warning about loss of modularity given previously applies here.

4 INSTRUMENTATION

The results of a simulation are primarily the insights it provides into the operation of the simulated system. The "insight" we frequently experienced using an early version of the simulation system was that tnore interesting results could have been produced by the run just completed if only the instrumentation had been different. With this in mind, the design for the current version of the simulation instrumentation system was aimed at flexibility. This was attained without significant performance impact by building efficient run-time system structures before each run, as outlined in section 1.1, from the declarations defining the instrumentation.

The organization of the instrumentation system is pictured in figure 5. The simulator interacts with component instances through assertions, that is, calls on an assert function, in behavior rules (the methods associated with : ApplyRul es messages). All instrumented components are specializations of an *instrumented-box* (as well as other classes). After each invocation of :ApplyRules for such components, the :ApplyRules method for a generic instrumented-box is appt ied. This causes invocation of the : trigger method for each *component-probe* associated with that component. Since this flow of measurements is accomplished by means invisible to the the writer of behavior methods for a component, the concerns surrounding component design are effectively partitioned from corn ponent instrumentation. The remainder of this section details these "invisible" means used to accomplish measurement flow during a simulation run as the measurements are staged from components through component probes to instrument panels.

4. I Component Probes

The first filtering of events is done by component probes. Some events cause no further measurement activity since, as it turns out, not all events merit action on the part of the instrumen tation system. The parameters of the event and the ports and state variables of the instrumented component dealing with the event are available to the component probe as are the state variables of the probe itself. Each piece of the selected information is tagged with an identifying keyword and passed along as the parameters of the : trigger method along with a keyword identifying the type of component probe, a number representing the current event time, and a pointer to the component with which the information is to be associated in the display. This pointer might be to some component related to the one actually handling the event, for example, the component enclosing it.

Component probes may he composed of predefined probe operation modules to do standard calculations (for example, moving averages) and then to forward the results to selected panels. In order to automate the composition of probes to accomplish such operations, each of these operations is chained together by invoking the method for that probe that is associated with

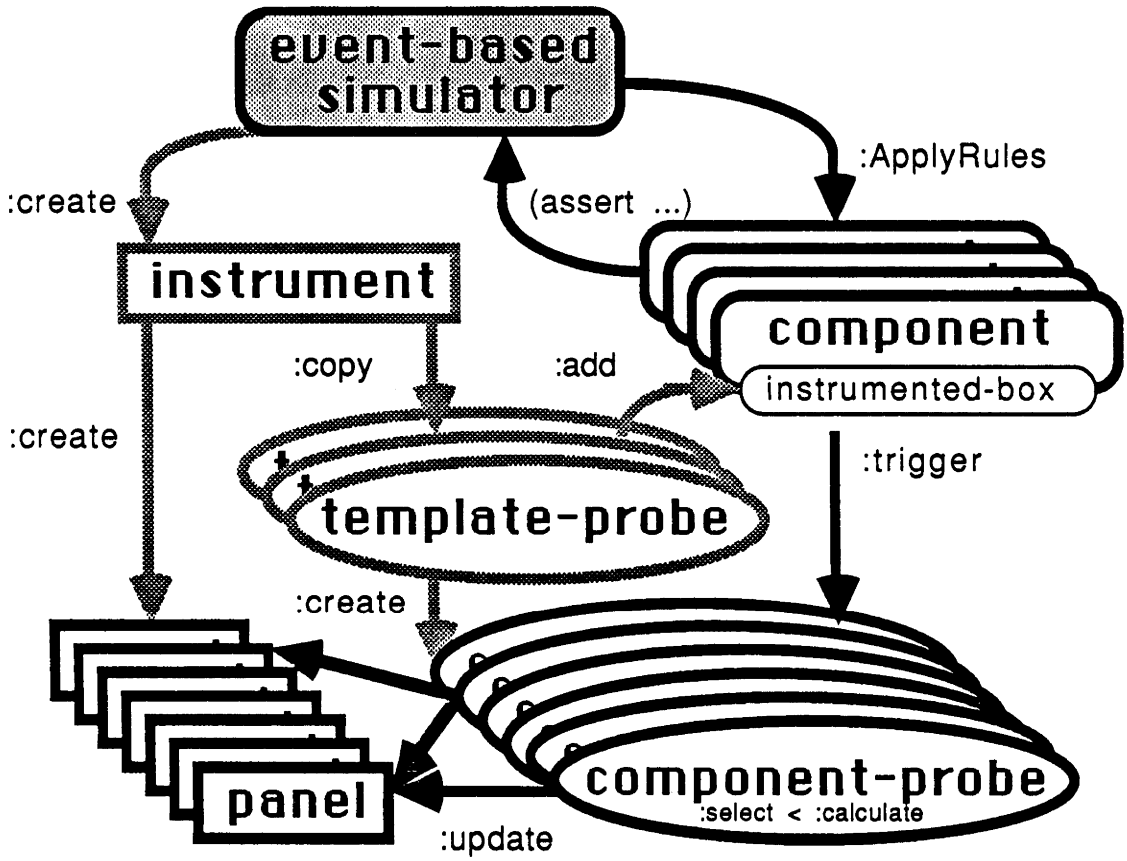

Figure 5: Instrument System Organization

the system-defined message name of the generic next operation. Thus, the : trigger method calls the : calcul ate method of the probe which, in turn, calls its : select method which, finally, calls the : update method of the selected panels associated with the probe. Probes are composed by naming them as specializations of appropriate probe operation modules (for example a : cal cul ate module for moving averages) as desired. The default, if no specializations are stipulated, is to pass through information without change to all the panels associated with a probe.

Information flow between components and panels is accomplished by the component probes associated with each instrumented component. The creation of such component probes and their association with appropriate components (by execution of : add methods) accomplishes the instrumentation of a circuit. This is done when an instrument is created. During simulation initialization, the components of the circuit (and their sub-components) to be instrumented are (recursively) examined by each *template probe* defined for the instrument to see if they are to be monitored. If so, the : copy method for the given template probe is invoked to create a new instance of the appropriate component probe and add it to the probes connected to the component. Each template probe previously received the identifiers for the panels to which its clones should send information. These will be the panels identified when a component probe invokes the : update method.

4.2 Instrument Specifications

The operations performed by an instrument panel are to:

- *Find* information previously stored according to the component pointer supplied by the : update method:

- *Link* new data structures as needed (to save such information) to other such structures of the panel;
- *Save* in these data structures the results of expressions that reference indicated keyed information from the : update parameters and the prior contents of the structures:
- *Send* the results of periodic analyses on the information associated with a panel for display by the same panel or by some other; and
- *Show* processed information in the manner specified for the panel.

The defaults for the panel operations supply the most commonly required specifications implicitly, so simple operations are simply specified. These defaults can be overridden as needed and either predefined or user specified alternatives for the panel operation's can be selected in their place. Arbitrarily complex (Lisp) expressions can be used to specify the transformations between the information provided by a probe and that saved and displayed by the panel.

These transformations and all the default overrides for the panel operations that are stipulated in the instrument declaration are scanned when a new instrument is created for a simulation session. They are compiled at that time into code bodies referenced by run time control blocks associated with each panel. A simulated system is instrumented by examining all of its components and attaching to each component the copies of template probes specified by the instrument definition that are appropriate for the component (by means of calls on the : copy and : add methods for the probe). This can be a many to many relationship as shown in figure 6.

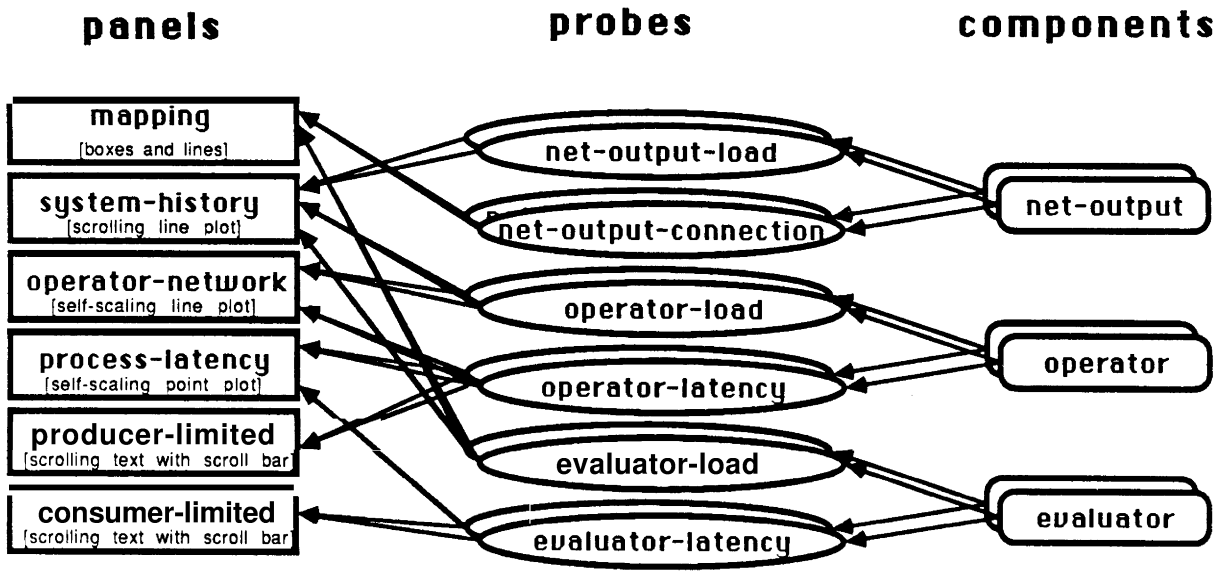

Figure 6: Instrument Probe and Panel Relationships

Component probes to measure "load" and "latency" are specified in the given example for each operator and evaluator in the circuit. The "load" and current "connection" for each net-output is also to be monitored. Some panels, for example the one showing "consumer-limited" processes, receive inputs from only one type of component probe, those measuring evaluator latency. Others, such as the one measuring "process-latency" receive inputs from more than one kind of probe (in this cast!, from probes measuring operator latency as well as those measuring evaluator latency). A way must thus be provided to distinguish the type of probe sending information to a panel: this is described in the next section.

Some probes send information to only one panel, for example, the net-output connection probes. Others monitor information which is needed by several panels, for example, the operator latency probe. Transformation of the raw information provided by a probe will need to be specialized to the information expected by each panel receiving it. A general way to stipulate these transformations is stipulated in the next section.

5 EXAMYLE PANELS

Some example panels are described in this section to give a feel for the instrumentation possibilities available in CARE and elaborate on how the requirements described in the previous section for probe type identification at a panel and per panel specialization of the information provided by a probe are handled.

5.1 Point Plot Panels

The first panel (shown in the left half of figure 7) is an example of a *point plot panel* used to generate a scatter plot. As an option, only points representing simulated activity over a limited past history from the most recent event time are kept for display. In this example, resource load[5] information is provided by the operator-load and evaluator-load component probes attached respectively to the operators and evaluators of the system.

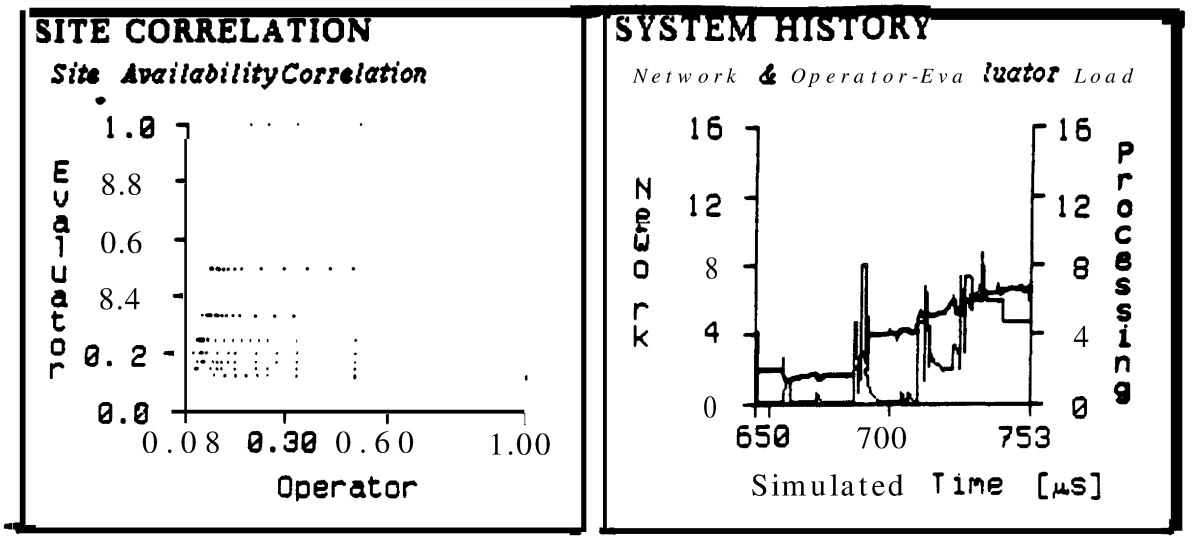

Figure 7: Point Plot and Scrolling Line Plot Panels

The balance between the "availability" of the evaluator and operator of each site, that is, the complements of their respective loads, is displayed during the simulation as events are processed that change this measure. In order to avoid capturing information at too fine a temporal granularity, previously gathered information for a given site is overwritten if it is within a given sampling interval of the new information. Information that is beyond a given history range is dropped. The scale of availabilities displayed is fixed between 0 and 1.0. The panel specification to declare all this and to also stipulate the axis labels is shown in figure 8.

[5]Resource load is defined as (1 - 1 / (1 + aggregate-queue-length)) where the aggregate queue-length is the sum of the lengths of all queues providing work for the resource.

```
'((("Operator") (0 1.0) (- 1 (:operator-load :busy)))      ;Bottom axis
  (("Evaluator") (0 1.0) ((- 1 (:evaluator-load :busy)))) ;Left axis
  :find (find-sample-distinct (:simulator :time) ,sampling-interval)
  :show (recent-history ( :simulator :time) ,point-panel-history-range 0))
```

Figure 8: Site Correlation Panel Specification

5.2 Scrolling Line Plot Panels

An example of a *scrolling line plot panel* is shown in the right half of figure 7. This panel sums the loads seen by the resources in the simulated system and displays this as a strip chart, the "system history". Some of the same probe load information used by the previous panel is used in this panel as well, but with different transformations defined in the panel specification as shown in figure 9.

```
'((("Simulated Time [us]") (,history-range) (:simulator :time))  ;Bottom
  (("Network") (0 ,sites) (:net-output-load :busy save-sum))     ;Left
  (("Processing") (0 ,sites)                                     ;Right
     (average (:evaluator-load :busy save-sum)
              (:operator-load :busy save-sum)))
  :find (update-history (:simulator :time) ,sampling-interval)
  :show (recent-history (:simulator :time) ,history-range 0))
```

Figure 9: System History Panel Specification

Line plot panels may have two independently scaled vertical axes. For the system history panel shown, the sum of network loads as indicated by the net-output components of the system is plotted against the left axis and the sum of the processing loads provided by the current-average of the sums of the operator and evaluator loads is plotted against the right axis. Event time is plotted on the horizontal axis. The update-history function uses the component pointer to find the information previously saved for that component and records the current event time as the (: simulator : time) so that it may be used to display information correctly on the horizontal axis. The current sums of the evaluator loads and the operator loads measured by the system are stored in a record for the given event time (or a prior event time within the specified sampling interval) by the calls to the save-sum function specified as part of the *save* operation.

5.3 Self Scaling Line Plot Panels

Figure 10 illustrates both the self scaling of displays and the use of a display analysis operation. For this self scaling line plot panel, two pieces of data are collected for each operator in the system: the load on the operator. shown on the right axis, and the latency of the information it has most recently received. This last item is provided by the operator latency probe in two parts: (1) the interval between the creation of the information and its receipt by the net-input feeding the operator and (2) the interval between such receipt and the operator taking action on it. There are thus two curves plotted on the left axis. The specification stipulates a list for the left axis display. The elements of this list are the "net delay" and the sum of this measure and the "operator delay" monitored by the operator latency probe. Since both delays are non-negative, their sum must be at least as large as either one taken alone: the two curves may be superimposed but can not cross. The difference between the two curves is the incremental delay added by the operator.

The panel specification for the operator-network panel is shown in figure 11. In addition to transformations shown previously, an analysis function is stipulated for the *send* operation of the panel. The information saved from each of the probes sending : update messages to the panel is to be sorted from the greatest to the least values of the associated sum of delays described above. This information is to be saved as the operator latency rank and used as such to determine the position on the horizontal axis that the delay and load information will be displayed.

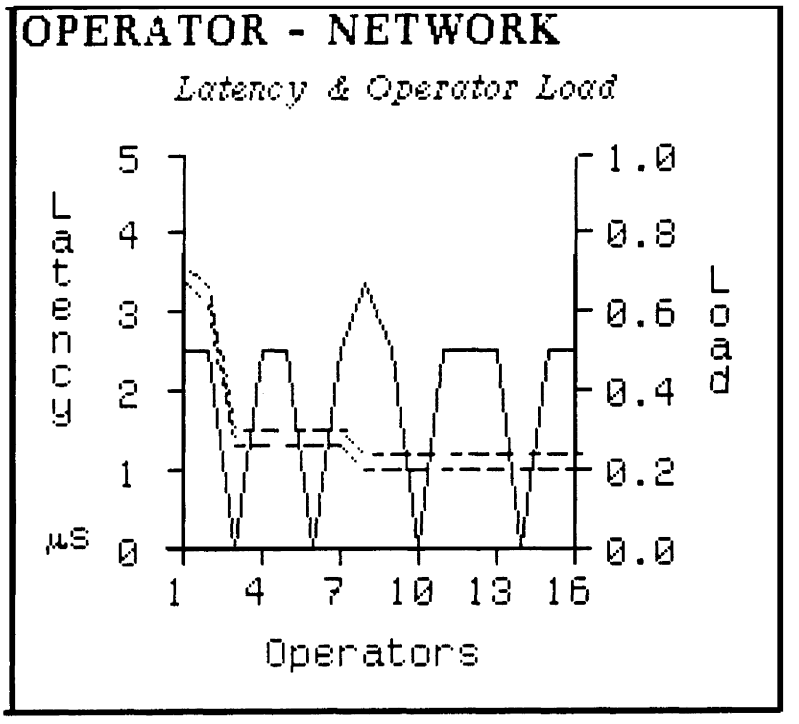

Figure IO: Self **Scaling Line** Plot Panel

```
' ( (("Operators") (1  ,sites) (:operator-latency :rank))
  (( ("Latency" "us")) (0 nil) ;Second string: 90 degree baseline shift
  `((:operator-latency (:net-delay (+ :net-delay :operator-delay)))))
  (("Load") (0 1.0) (:operator-load :busy))
  :send (sort-arrays
          ((,#'> (:operator-latency (+ :net-delay :operator-delay))))
          ((:operator-latency :rank))))
```

Figure I I: Operator-Network Panel Specification

5.4 Boxes and Lines Panels

Perhaps the most intuitively satisfying of the types of panels available is the *boxes and lines panel,* a graphic representation of a circuit showing its components and their interconnections. An example of such a panel is shown the left part of figure 12. This class of panels uses information left behind by the structure editor when the circuit was defined. Its form is thus automatically generated. The position of the components ("boxes") and the connections between them ("lines") in the display are used to animate system operation. In the example shown, the shading (or color) of the boxes is used to indicate the availability of the *evaluators* in the simulated system as the simulation proceeds. Darkest shades indicate highest availability, that is, empty queues for utilization of the resource; lighter shades indicate lower availability, that is, longer queues. The lines between boxes indicate communication paths that are in use, that is, not ": free" at the time of the most recent *show* operation for the panel.

The panel specification for the *mapping panel*, an instance of a boxes and lines panel, is shown in figure 13. There are two specifications for the panel: one for the boxes and one for the lines. The specification for boxes in the panel stipulates that the availability of evaluators in the sites corresponding to the boxes displayed controls the shading of those boxes. The scale is defined to run from 0 to 1.0. The specification for lines in the panel uses the connection information reported for the net-output to determine line placement on the display. When the status is reported as `:free`, the connection information is dropped from the panel and the corresponding lines are removed.

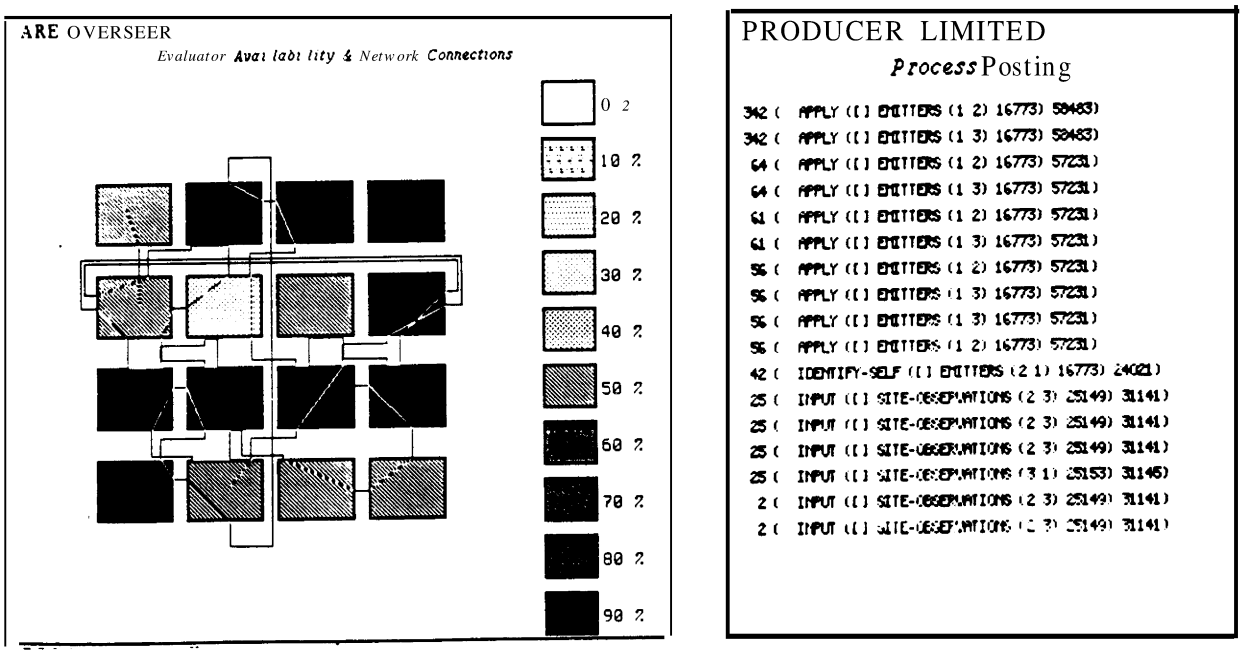

Figure 12: Boxes and Lines Panel and Scrolling Text Panel

```
'((("Evaluator Available") (0 1.0) (- 1 (:evaluator-load :busy))))
'((("Packet Trace") nil (:net-output-connection :points))
 (("Packet Status") nil (:net-output-connection :status))
  :find (find-and-remove ,#'eq (:net-output-connection :status) :free)))
```

Figure 13: Mapping Panel Specification

5.5 Scrolling Text Panels

Sometimes, the most appropriate way to display information is to show it as text. Based on a similar facility provided by the underlying Lisp system, the *scrolling text panel* provides a scrollable window into lines of text. In the right part of figure 12, the delay in each process execution while waiting for something to do, that is. the eventtime interval spentwaiting for an appropriate task to appear on a certain stream of tasks, is shown together with the process that finally produced the awaited work. This information is sorted so that the text lines appear from the greatest stream waiting interval to the least.

```
'(( ( ) ("~4D - A")
   ((fix (:stream-waiting :interval))    ;first field
    (let* ((origins (packet-origin (:stream-waiting :packet)))
           (origin (if (listp origins) (first origins) origins)))
      (remote-address-local origin))))  ;second field
  :send (sort-arrays ((,#'> (:stream-waiting :interval))) nil))
```

Figure 14: Producer Limited Process Panel Specification

The values and formats used for display in a scrolling text panel are defined much as in previously defined panels. Format control strings take the place of scale information. As usual, values are described by a list of forms, each one of which specifies the transformations to perform on information received from probes. The example specification in figure 14 shows the generality with which probe information can be incorporated in Lisp expressions

to produce transformation specifications. The information used to generate the value for the second field of the text display is based on the origin of the task packet that arrived on the stream the process was waiting for.

5.6 Noting Simulation Parameters

The CARE component models are parameterized through menu interaction as shown in figure 15 to allow easy variation of their performance characteristics relative to each other. Additionally, the site model parameterizes alternative routing strategies: *directed,* that is, blocking when progress can not be made toward the goal; *spiraling* around the goal if progress toward it is blocked; and *dithering,* that is, routing away from the goal even if only the last link towards it remains to be acquired. The rate at which each site accepts application data is also a parameter, the *data rate* and can be used by an application to control how hard it drives the simulated system.

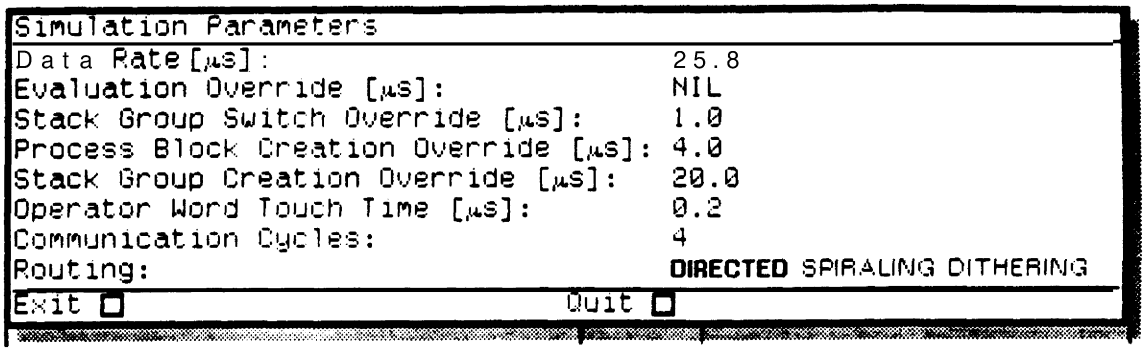

Figure 15: Parameter Menu

Many of the CARE parameters are specified as *overrides.* If not specified, the corresponding performance is taken as measured on the simulation machine. Thus, the *evaluation override,* that is, the time to perform an evaluation can be specified as non-nil in order to fix the time that each user evaluation will take. (This is useful in making runs repeatable for debugging). The time that it takes to switch context can he specified *as* the *stack group switch override.* Similarly, the time to create a process control block and a stack context for that process can be taken as given rather than measured by specifying respectively the *process block creation override* and the *stuck group creation override.*

The time required for operator execution is modeled in terms of the number of words the operator must manipulate in handl i ng a given message, The manipulation time per word is specified by the *operator word touch time.* Lastly, the performance of the communication subsystem is specified *as communication cycles.* This is done in terms of the minimum number of evaluator data path clock times (that is, event times) required for a 32-bit word to pass a given point in the network. Thus the parametric specification, "4 communication cycles", dictates that 8 bits may cross such a boundary each time the evaluator passes through one event time. If the communications path were narrower or the base communication clock rate were lower, a higher number would be specified.

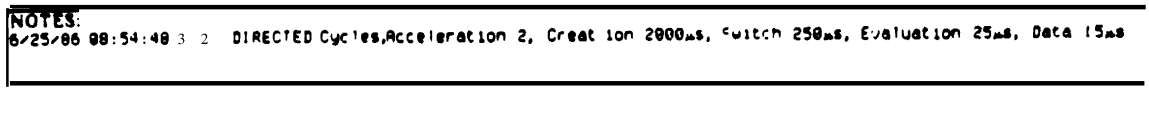

Figure 16: Annotation Panel

The last example of SIMPLE panels is the annotation panel as illustrated in figure 16. This

is used to (automatically) record the date, time, and parameters of the simulation run as well as any other information the user chooses to keyboard into it.

5.7 An Instrument Screen

All these panels are put together in an instrument screen according to a set of layout constraints manipulated by the underlying window system. The finished screen might look like figure 17. The instrument screen is redrawn at a rate set by the user. By experience, it is often better to update the screen at a frequency low enough to let the user interpret each screen comfortably than at the maximum rate possible. This approach also restricts the computing resources consumed by the instrumentation system. More focused approaches to controlling instrumentation load on the system include the ability to freeze selected panels and disconnect selected probes during a simulation run.

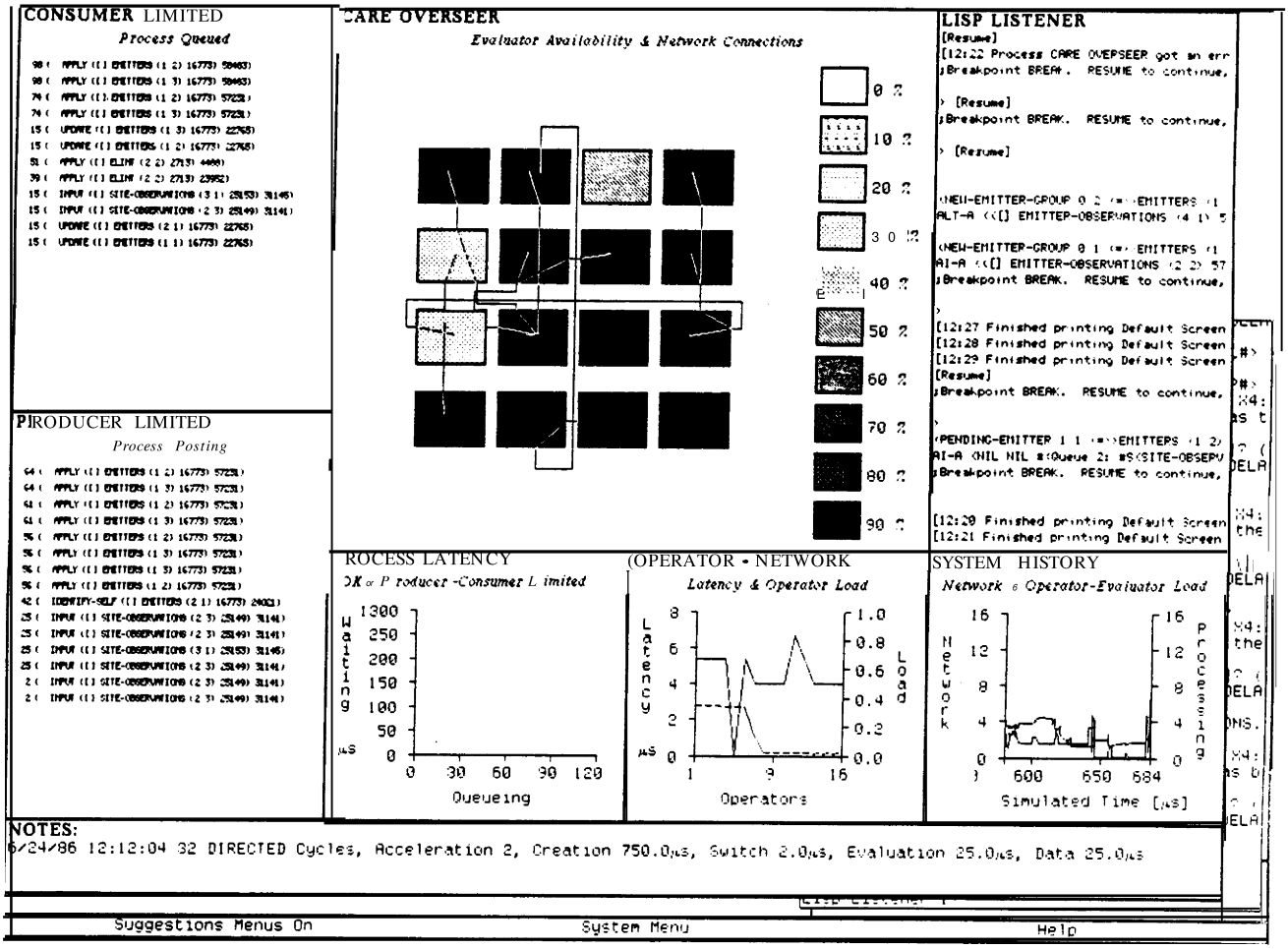

Figure 17: Overseer Instrument

6 USING YROGRAM DEVELOPMENT TOOLS

The SIMPLE/CARE simulation system is integrated into the underlying Lisp machine program development environment. The objects and data structures at both the component model and application language interface have abstraction interfaces that provide summary

state information when they are displayed in test forni. These text abstractions are "mouse sensitive" in the development machine environment and so can be inspected at successively finer levels of detail as desired.

In figure 18, the net-output components of the site at grid coordinates (3 2), the particulars of the net-output on the east side of the site (that is, net-output-3), and a summary of all the sub-components of the site at (3 2) are being inspected. This same kind of view into the progress of a simulation is provided in the debugging process and may, as shown in figure 19, refer to the conceptual entities of the application that is driving the simulated system.

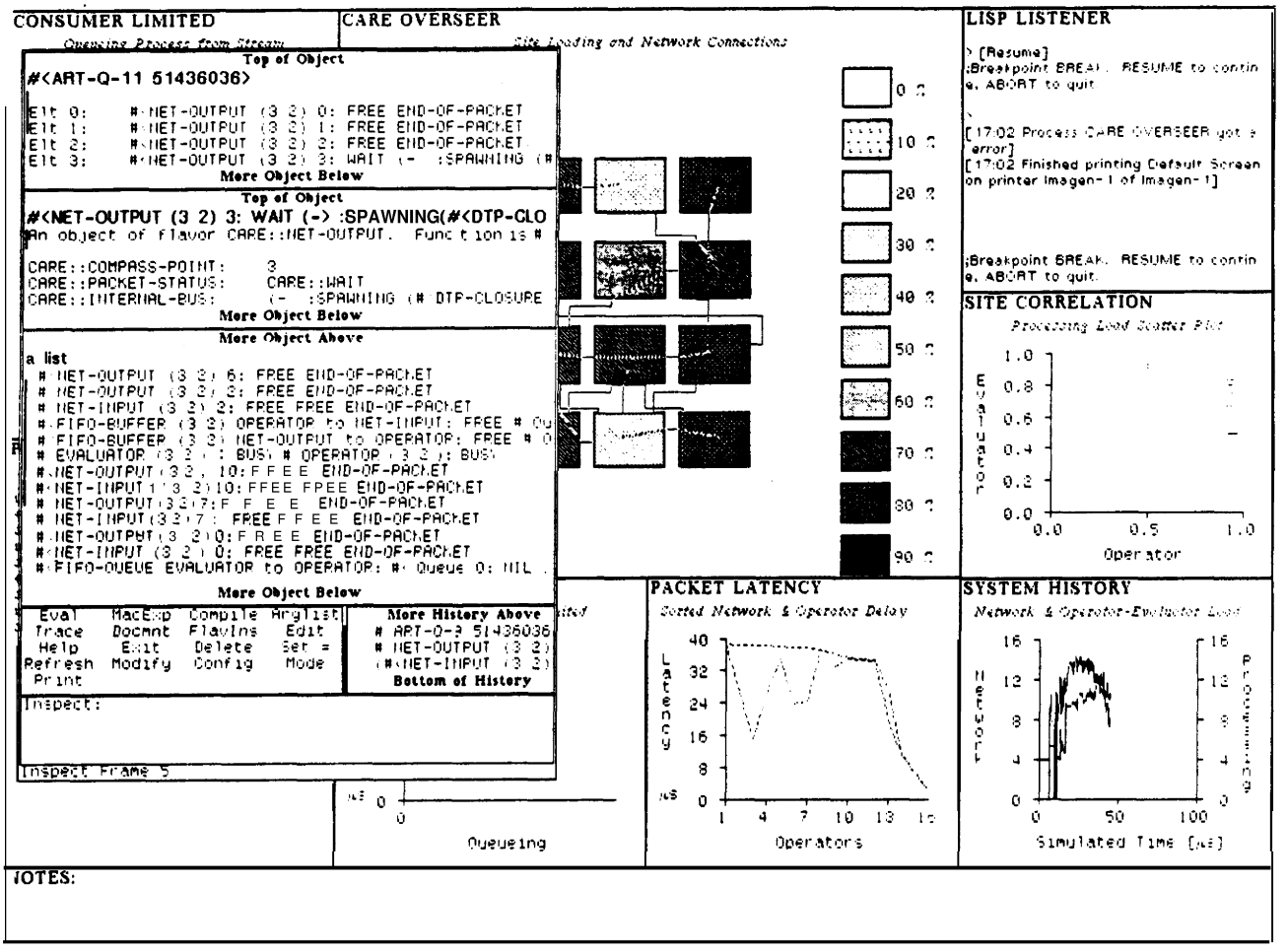

Figure 18: Inspecting Simulated Components

fn the example shown in figure 19, a distributer process running on the evaluator at site (1 1) has made an improper call on the update-locale function during execution of its **:start** method. It might have been appropriate to investigate this situation in terms of the modeled components. That could be done, for example, using the debugger to inspect the evaluator component, its enclosing site, related net-output components, or whatever else at the component model level seemed relevant. In this case, what was done was to use a few mouse clicks to indicate interest in the source file for the distributer : start method generating the problem. It was brought up for review and control was then transferred to an editor using the underlying program development environment as shown in figure 20.

Because of the implementation system chosen for the realization of SIMPLE/CARE, at any point in the simulation, procedures either in the application or in the component models can be modified, incrementally recompiled (within a few seconds), and be made effective for all

calls on them -- even those in the interrupted stack frame. Thus simulation execution can be backed up to some previous point in the stack frame and retried (given that intermediate side effecting code, if any, is safely re-executable).

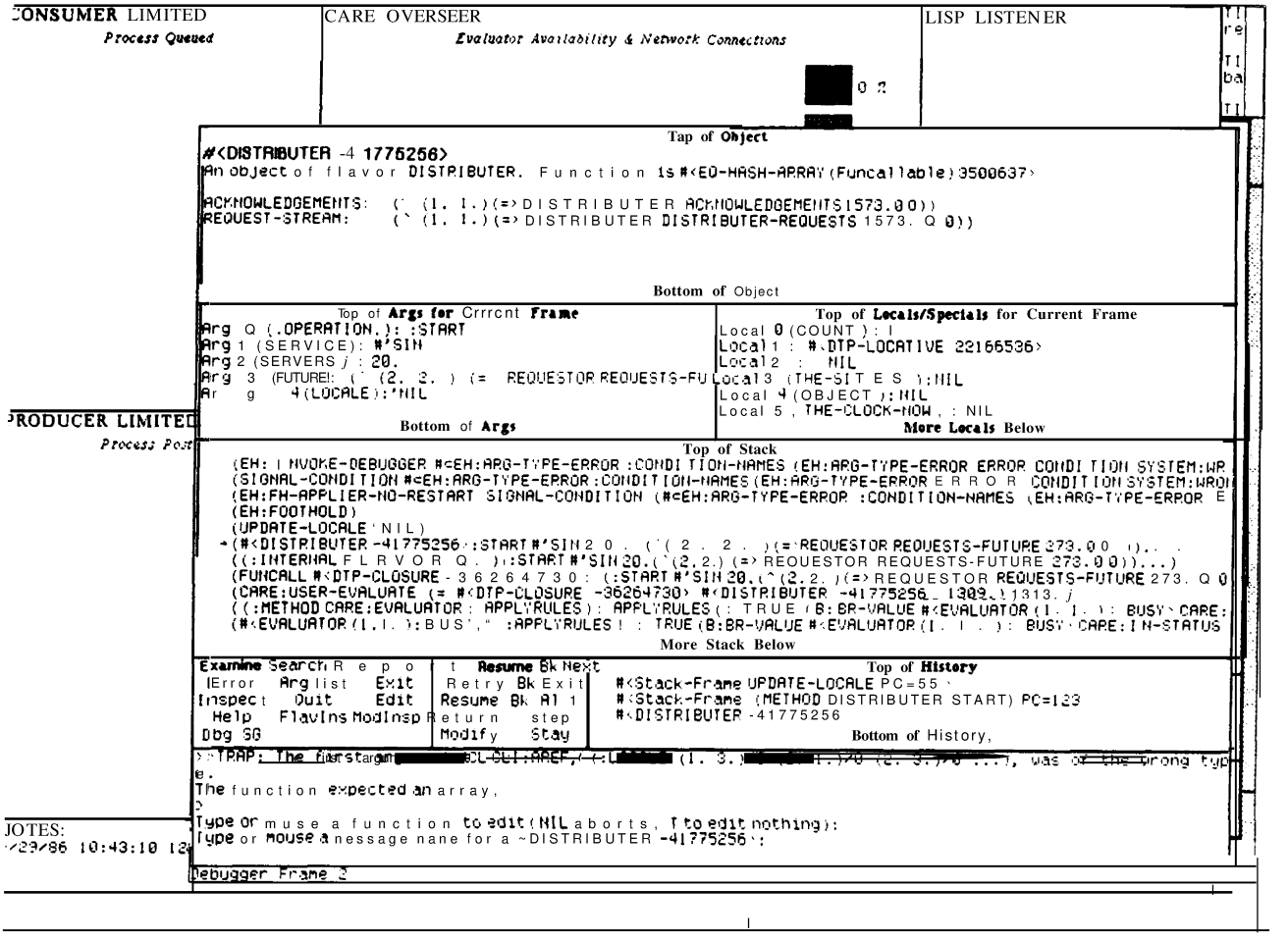

Figure 19: Debugging A Simulation

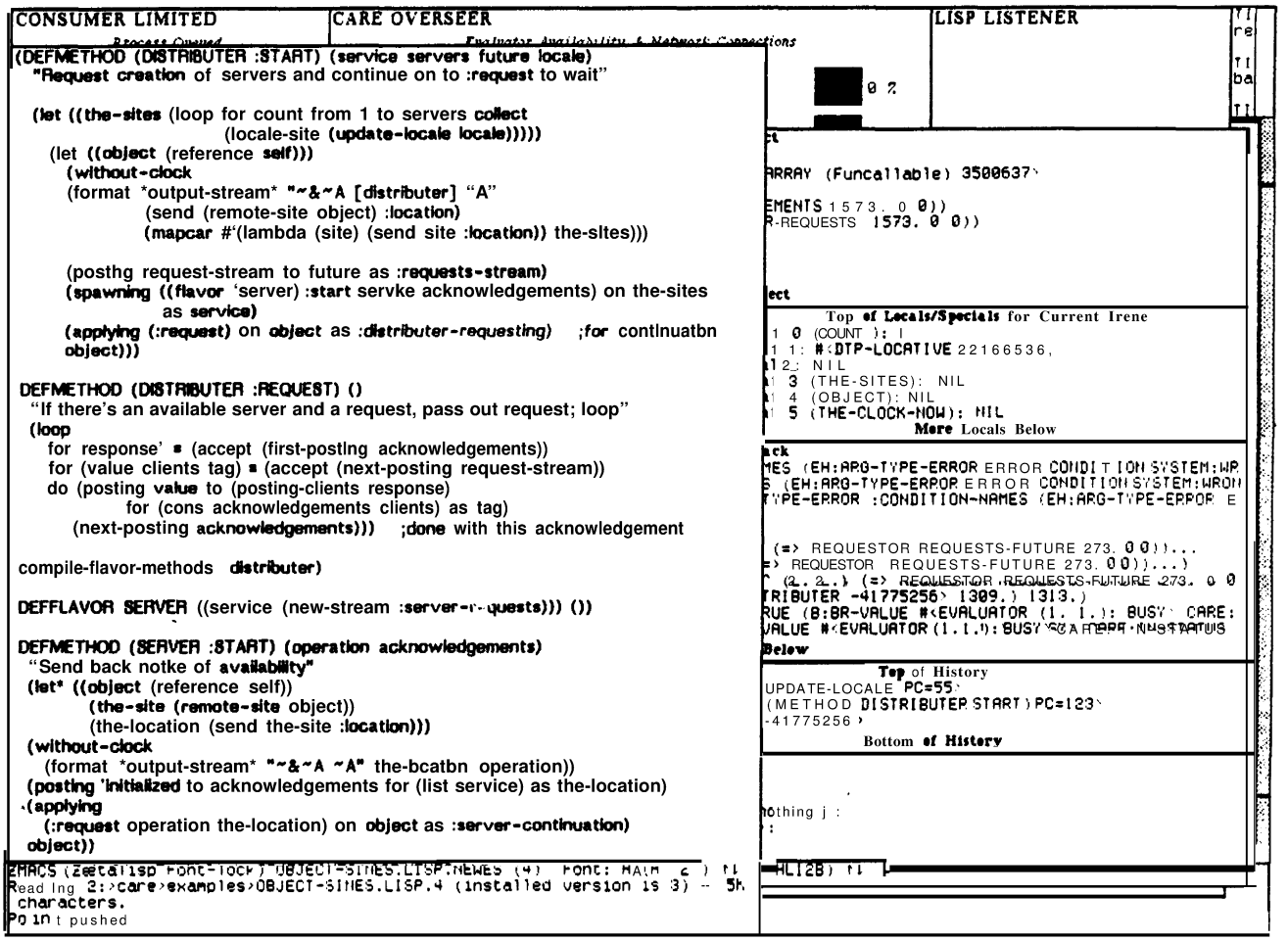

Figure 20: Changing Application Code

7 CONCLUSIONS

The goals of simulation flexibility and simulation environment completeness have been dealt with in the ways described throughout this paper. In summary, the system is flexible in that it supports:

- Arbitrary data types and lengths in simulation. The information whose flow and creation is controlled by simulated components may be of arbitrary complexity -- from numbers and keywords to procedure bodies and execution environments.
- Instantaneous effect of definition change at both the application and component modeling level (even during a simulation run).
- A broad range of instrumentation customization. Customizations may involve arbitrary expressions for probe data transformations, many to many probe to panel mappings, information from summary analyses on one panel's data included in another, and control of what state is saved and for how long.
- Separation of probe and component definitions to facilitate their independent modification.
- An application language interface that is easily extended or changed without recasting the information flow control described by the component behaviors.

While there is always room for additional capability[6], SIMPLE/CARE is a usefully complete system. It now includes:

- Supplied components for a network multiprocessor simulation with many of their parameters customizable by menu interactions.
- A- hierarchical structure editor that currently provides automatic grid and torus coniposi tion operators. (Automated composition of richer topologies, such as hypercubes, has been provided for in the basic design).
- A rule language that supports a synchronous design style without incurring the overhead of (naive) synchronous simulation.
- Method invocation for functional simulation that is integrated into the behavioral simulation rule system and which provides for operations by and on both local and hierarchically related components.
- Method specification design aids provided by the underlying program development environment (for example, method dictionaries and quick access to method sources from the debugging system).
- An evolved set of panel templates providing sorted, scrollable text lines as well as self and fixed scaling, "two and a half" dimensioned, history sensitive displays which may be scatter plots, strip charts, line graphs, intensity maps, and signal animations.

We set off to build a multiprocessor simulation system with performance adequate for the understanding of multiprocessor systems executing significant applications. The SIMPLE/CARE simulation system has been used to study the operation of "expert systems" of respectable size [2]. Depending on instrumentation load, these studies have involved simulation runs from 20 minutes to several hours each. While faster would surely be better, performance has proven adequate to these needs.

[6] A *histogram panel*, for example, is just now being added to the system

8 ACKNOWLEDGEMENTS

This work stands on the shoulders of its predecessor, the Palladio system, designed and implemented by Harold Brown and Gordon Foyster. Our functional goals were more restrictive than theirs so we had the luxury of design by simplification. Without their implementation base, it would have been hard to know even where to begin.

Many hands and minds have contributed to the development of SIMPLE/CARE. We are particularly indebted to the work of Russ Nakano who started off to do a simple learning exercise and ended up doing a particularly careful modeling of a intricate signalling protocol.

References

1. Brown, Harold, Christopher Tong, and Gordon Foyster. "PALLADIO: An Exploratory Design Environment for Integrated Circuits." IEEE *Computer 16* (December 1983).

2. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Tech. Rept. STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

3. Greg Byrd, Russell Nakano, and Bruce Delagi. A Point-to-Point Multicast Communications Protocol. Tech. Rept. KSL-87-02, Knowledge Systems Laboratory, Stanford University, January, 1987.

4. Daniel Weinreb and David Moon. *Lisp Machine Manual.* Symbolics, Cambridge, MA, 1981.