November 1987

Report No. STAN-CS-87-1189
Also Numbered KSL-87-65

Instrumented Architectural Simulation

by

B. A. Delagi, N. Saraiya, S. Nishimura, and G. Byrd

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL 87-65

November 1987

Instrumented Architectural Simulation

by

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

Digital Equipment Corporation
Maynard, Massachusetts 01754

Stanford University
Stanford, California 94305

This work was supported by DARPA Contract ***F30602-85-C-0012,*** *NASA Ames Contract NCC* ***2-220-S1,*** *and* ***3oeing*** *Contract* ***W266875.*** *Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University, Department of Electrical Engineering.*

Instrumented Architectural Simulation

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

Digital Equipment Corporation
Maynard, Massachusetts 01754

Stanford University
Stanford, California 94305

ABSTRACT

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? 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].

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]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 [3] and currently runs on Symbolics 3600 machines and TI Explorers.

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.

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 defined 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 initialized, 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 genera\ *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.

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.

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]

[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 primitives.

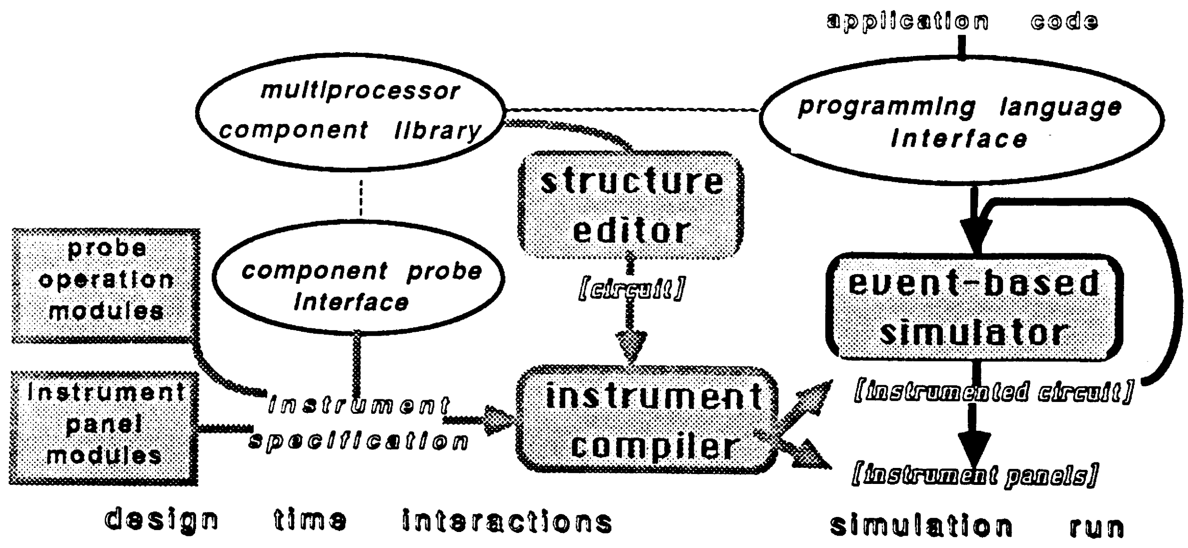


Figure 1: Design Time Interactions and Run Time Representations

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 and so on indefinitely until the top level "circuit", the system structure, is reached.

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.

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 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 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 more 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.

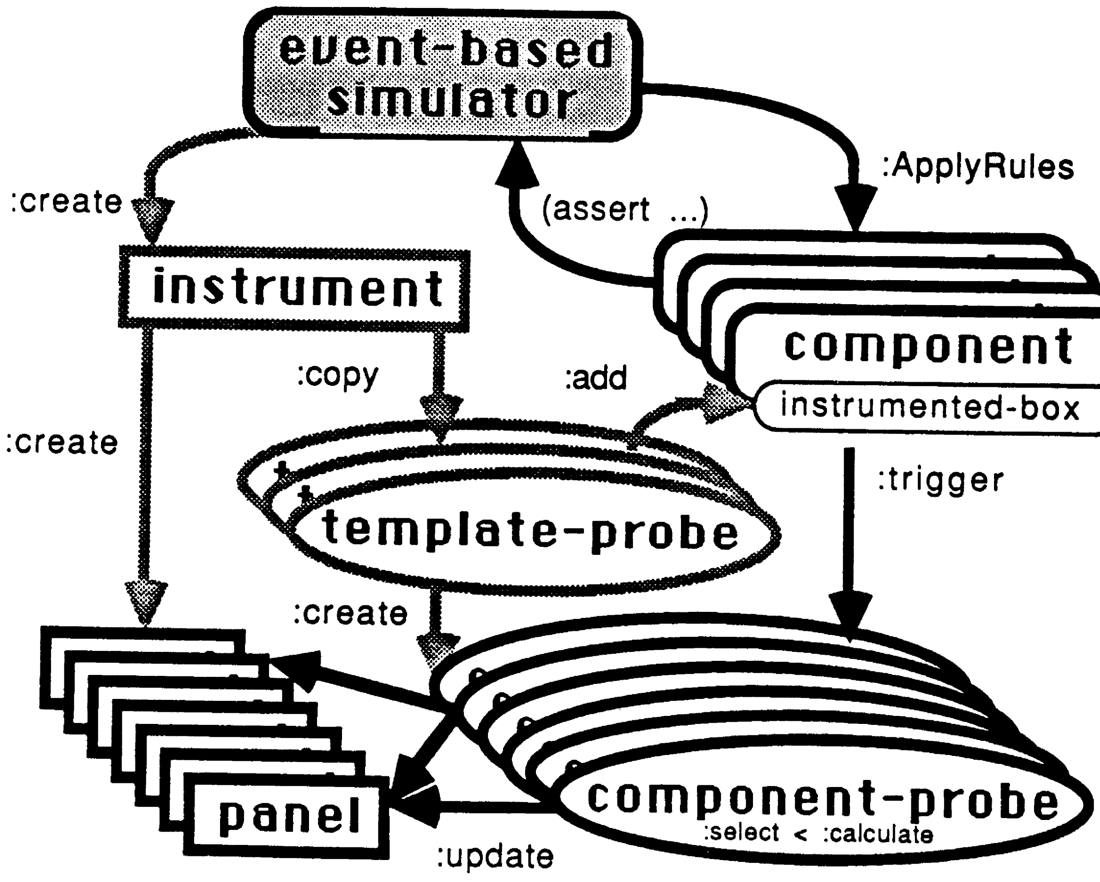
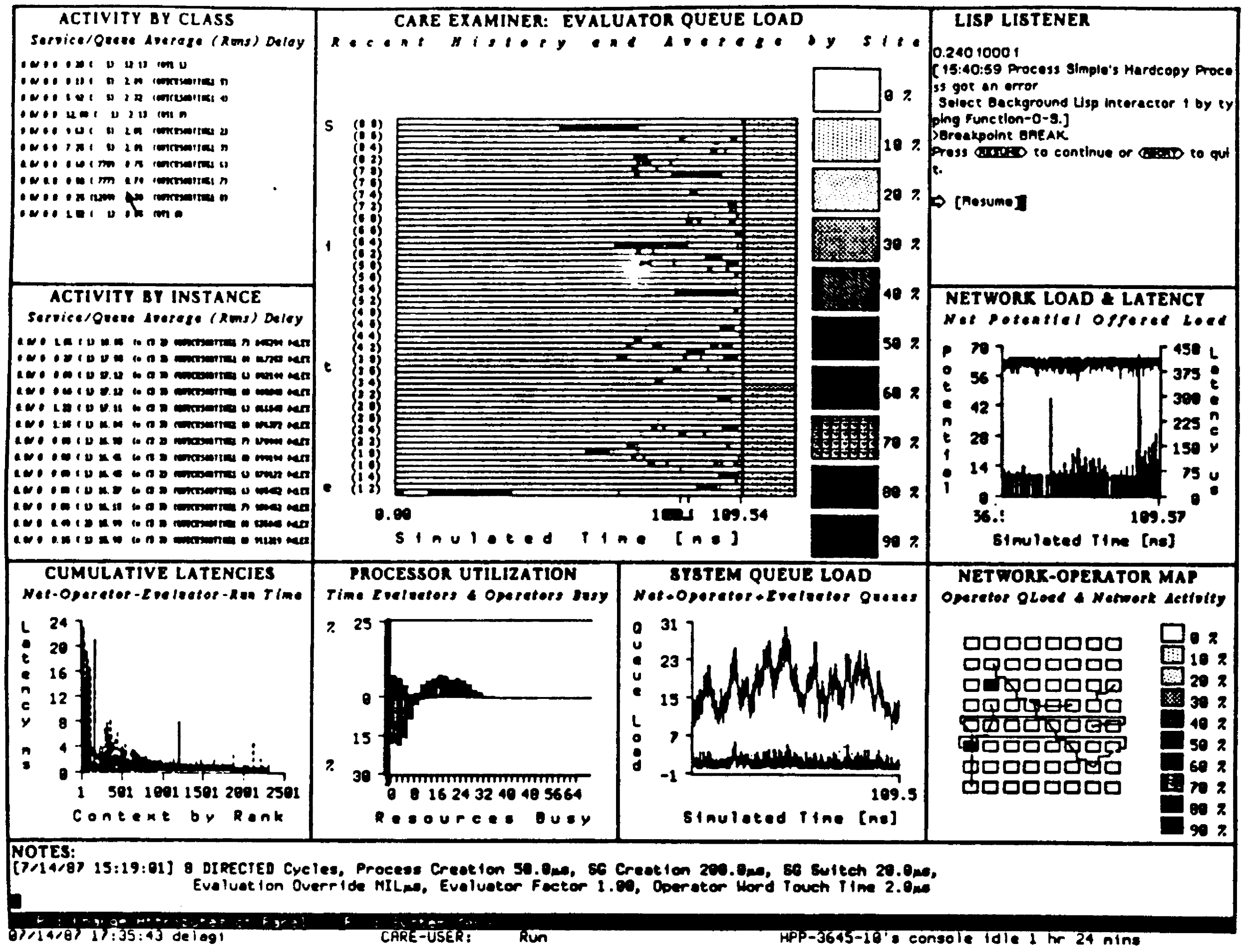


Figure 2: Instrument System Organization

The organization of the instrumentation system is pictured in figure 2. The simulator interacts with component instances through assertions, that is, calls on an assert function, in behavior rules (the methods associated with **:ApplyRules** 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 applied. This causes invocation of the `:trigger` method for each *component-probe* associated with that component. Data from component probes is collected and displayed by *instrument panels.* 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 component instrumentation. 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 3.



```
0.240 1000 1
[15:40:59 Process Simple's Hardcopy Proce
ss got an error
 Select Background Lisp interactor 1 by ty
ping Function-0-S.]
>Breakpoint BREAK.
Press [illegible] to continue or [illegible] to qui
t.

⇨ [Resume]
```

```
NOTES:
[7/14/87 15:19:01] 8 DIRECTED Cycles, Process Creation 50.0µs, SG Creation 200.0µs, SG Switch 20.0µs,
                  Evaluation Override NILµs, Evaluator Factor 1.00, Operator Word Touch Time 2.0µs

07/14/87 17:35:43 delagi          CARE-USER:     Run          HPP-3645-10's console idle 1 hr 24 mins
```

Figure 3: Overseer Instrument

4 CONCLUSIONS

The design goals of simulation flexibility and simulation environment completeness have been supported as discussed above. 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 inf rmation flow control described by the component behaviors.

While there is always room for additional capability, 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 composition 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 histograms and 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.

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