Report 83-31
Stanford -- KSL

Scientific DataLink

PALLADIO: An Exploratory Environment for Circuit Design.
Harold Brown, Christopher Tong, Gordon Foyster, Dec 1983

card 1 of 1

June 1983
Also numbered: HPP-83-31

Report No. STAN-CS-83-984

Palladio: An Exploratory Environment for Circuit Design

by

Harold Brown, Christopher Tong
and Gordon Foyster

Department of Computer Science

Stanford University
Stanford, CA 94305

HPP 83-31
STAN-CS-83-984

Reprinted, with permission, from Computer, Vol.16, No.12, pp.41-56, December 1983.

The experimental Palladio environment recognizes the need to integrate tools and languages in an attempt to create a flexible design framework.

Palladio: An Exploratory Environment for Circuit Design

Harold Brown, Christopher Tong, and Gordon Foyster, Stanford University

Palladio* is a circuit design environment for experimenting with methodologies and knowledge-based, expert-system design aids. Its framework is based on several premises about circuit design: (1) circuit design is a process of incremental refinement; (2) it is an exploratory process in which design specifications and design goals coevolve; and (3) most important, circuit designers need an integrated design environment that provides compatible design tools ranging from simulators to layout generators, that permits specification of digital systems in compatible languages ranging anywhere from architectural to layout, and includes the means for explicitly representing, constructing, and testing such design tools and languages.

The Palladio environment is part of a growing trend toward creating integrated design environments and away from isolated design aids. Recently several commercial computer-aided engineering workstations have emerged,[1] providing multiple-level, circuit-specification entry systems and integrated analysis aids. Integrated circuit designers have a special need for such workstations because of the complexity of large integrated circuits and the high costs of prototyping them.

Integration eliminates—or at least reduces—many problems that exist only because of unnecessary information loss. For example, circuit extraction programs are required, in part, because one stage of the design process—circuit design—fails to communicate its results to another stage—layout generation. Designers often make long chains of decisions, only to revise them when the disadvantages become apparent. They may, for example, implement a multiplexer component with random gate logic, then retract that solution when a PLA implementation appears more area-efficient. The lag between the original implementation and analysis of its consequences is, in part, inherent in the exploratory nature of design; however, some of the delay can be eliminated in an environment in which designers can describe known laws, heuristics, and trade-offs connecting high-level decisions (architectural decisions) with low-level consequences (area of a layout). The Palladio environment integrates both design tools and design specification languages and provides the conceptual framework required by such integration. That such integration has been delayed until now reflects the complexity of the underlying design knowledge.

Palladio differs from other integrated design environments by providing the means for combining, constructing, testing, and incrementally modifying or augmenting design tools and languages. For the circuit designer, it supports the construction of new specification languages particular to the design task at hand and supports augmentation of the system's expert knowledge to reflect current design constraints and goals. For the design environment builder, it provides several programming paradigms: rule-based (expert-systems oriented),[2,3] data-oriented, object-oriented, and logical-reasoning based. These capabilities are largely provided by two of the experimental programming environments in which Palladio is implemented: LOOPS[4] and MRS.[5] Specific features of these paradigms make them useful for constructing different elements of a design environment.

Perhaps the most significant property shared by these programming paradigms is their ability to represent ex-

*Andrea Palladio (†1580) was the Italian architect who developed the proportions and formal architectural style known as classical architecture. In a sense, he was the first knowledge engineer of design principles, and four hundred years after his death, his influential works are still in print.

0018-9162/83/1200-0041$1.00 © 1983 IEEE

plicitly specific kinds of design knowledge: a representation is explicit if it can be treated and manipulated as data by a program. Much artificial intelligence research in the last 10 years can be viewed as contributing to a classification of different kinds of knowledge and providing explicit knowledge representations for each kind.[6] Designing an integrated circuit requires a significant body of expert knowledge with a wide variety of forms, often technology-specific or even circuit-type specific. The expert knowledge contained in a design environment should be accessible and understandable to a designer. Much valuable expert knowledge can take the form of a well-integrated collection of task-specific design aids. One of the primary functions of Palladio is to be a vehicle for acquiring and recording circuit-design knowledge.

Palladio provides a testbed for investigating elements of circuit design that includes specification, simulation, expert-system design aids, and use of previous designs in a current design. It has facilities for conveniently defining models of circuit structure or behavior. These circuit models, called perspectives, are similar to circuit design levels; the designer can use them interactively to create and refine circuit design specifications. Perspectives can include composition rules that constrain how circuit components may be combined to form more complex components.

Circuit design can be viewed as the transformation of an initial specification into a final one that adequately details the circuit.

This integrated design environment provides menu-driven, graphics interfaces for editing and displaying structural perspectives of circuits in a uniform manner and a uniform behavioral language with an associated behavioral editor for specifying a design from a behavioral perspective. Further, a generic, event-driven behavioral simulator can simulate a circuit specified from any behavioral perspective and can perform hierarchical and mixed-perspective simulation. A color graphics display can be used for showing dynamic simulation "movies," and the environment offers several facilities for implementing design refinement aids as expert systems and for conveniently creating and using libraries of prototype components. These libraries can contain components of arbitrary complexity.

The circuit design process

Circuit design can be viewed as the transformation of an initial specification into a final one that adequately details how the circuit is to be built. The initial specification, which is usually imprecise and incomplete, focuses primarily on the circuit's functionality. The final specification emphasizes the structure, for example the geometry of the fabrication masks, but can also include specifications of desired circuit behavior and qualitative performance. Unless the circuit is so simple that its full detail can be grasped at once, the transformation of an initial specification into one that can be built is an incremental process during which abstract specifications of the circuit's structure and behavior are gradually refined into more concrete specifications. The evolving design specifications must be tested and evaluated against design goals and constraints. The circuit design environment should provide tools to assist the designer in both design specification and design verification throughout the refinement process.

Circuit design resembles in many ways that of complex software systems,[7] but it resembles exploratory programming more than structured programming. An exploratory circuit design environment, like an exploratory programming environment, must provide an integrated set of tools to assist the designer. We have modeled our circuit design environment after the Lisp programming environments developed for artificial intelligence research, for example Interlisp.[8] In these environments, the design of a program (i.e., the code) and of the program's specifications can evolve together, since the programmer has a well-integrated set of powerful tools to assist in the development process (e.g., file management, interactive graphics, and debugging tools).[9]

Hierarchical design. Many factors complicate design refinement. The space of refinements is large and its elements are complex, the generation and evaluation of a refinement is expensive, and only partial information is available at any step in the process. Furthermore, it is impossible to predict all of the consequences of choosing a particular refinement, so designers often retract one refinement and pursue another.

Designers have, in part, coped with the difficulties of the design process by using the principle of divide and conquer. As Simon[10] (among others) has observed:

> To design . . . a complex structure, one powerful technique is to discover viable ways of decomposing it into semi-independent components corresponding to its many functional parts. The design of each component can then be carried out with some degree of independence of the design of others.

This top-down technique of hierarchical partitioning is used universally by designers to simplify the design process. It is in itself a difficult and knowledge-intensive task. An inappropriate partition simply transfers complexity from the process of designing the constituent components to the subsequent process of recomposing the designed components into an overall design. Appropriate design specification levels help ensure that recomposition is manageable.

The design paradigm supported by Palladio is an incremental refinement of design specifications, with periodic validation of the specifications by simulation. The basic hierarchical step in structural refinement is to partition components specified at a given structural description level (i.e., from a given perspective) into constituent components specified either at that level or at a less abstract level. In addition, detail can be added to the structural specification of a component (e.g., clock phase to a register or mask level to a wire) and behavior can be specified in various behavioral description languages. The hierarchical use of multiple structural and behavioral descriptions and the allowed refinement steps constitute Palladio's model of the design process.

Design specification

In Palladio, designs are specified with design perspectives. A design perspective provides a conceptual model for viewing the structure or the behavior of a circuit that emphasizes certain features while suppressing others, as well as a language for specifying the circuit from the perspective. For example, a circuit can be viewed as a finite state machine or as a collection of clocked storage registers and combinational logic elements.

A Palladio perspective is either structural or behavioral. This explicit decoupling of behavioral perspectives from structural perspectives allows a modularity not admitted by most circuit design languages. Within Palladio, one or more behavioral perspectives can be associated with a structural perspective and vice versa. This permits, for example, a component to be specified from a very abstract, behavioral perspective at an early stage in the design process, then from a more concrete, behavioral perspective later in the process. Moreover, a circuit designer can use Palladio to construct structural and behavioral perspectives specifically tailored for a particular circuit. Examples of such circuit-specific perspectives appear later in this section.

Palladio defines a structural perspective by the types of components allowed when partitioning a component into constituent components with respect to that perspective. For example, from a register and combinational logic perspective, the only types of components allowed are subsystems, registers, combinational logic blocks, and interconnect. Some of the component types of a perspective are primitive with respect to the perspective: they cannot be partitioned from the perspective. Other component types are composite with respect to the perspective: they permit further decomposition from the perspective. For example, from a switches-and-gates perspective, a component of type switch is primitive while a component of type gate may be further partitioned into switches and other primitive components. A circuit is fully partitioned in relation to a perspective if all components at the lowest level of the circuit's component hierarchy are of primitive types.

The definition of a perspective can also include composition rules that limit the ways in which components can be interconnected. These rules help ensure that circuits specified from a perspective are correct with respect to that perspective's concerns. The allowed component types and the composition rules of a perspective constrain the manner in which a given component can be partitioned with respect to the perspective. In this manner, the use of structural perspectives complements the component decomposition process and helps ensure that the recomposition process is manageable.

Examples of structural perspectives. We now present two examples of experimental perspectives that have been implemented using Palladio. The first example is useful for designing a broad variety of NMOS circuits. The second example was constructed to investigate a specific machine architecture.

Example 1: CSG perspective. One of the circuit-level perspectives that we have implemented is the clocked switches and gates. From the CSG perspective, the designer views a circuit as networks of steering logic, clocking logic, and restoring logic gates. It is specialized for NMOS circuits that use a two-phase, nonoverlapping clocking scheme. It is concerned primarily with the circuit's digital behavior, that is, with the avoidance of indeterminate logic levels. Such indeterminate levels can be caused by improper connection of components, improper operating regions of devices, and leakage of stored charge. The CSG perspective is based on concepts presented in Stefik[11] and is closely related to switch level simulators.[12]

The primitive component type for steering logic is a steering switch, which represents a pass transistor with ports labeled as shown in Figure 1a. The composite component type steering net is used to represent networks of steering switches. An example of a component of type steering net, a 2-by-2 barrel shifter, is shown in Figure 2a. As their names imply, steering switches and nets are used solely for guiding data—they perform no clocking or data-storage functions.

The primitive component type for clocking logic is a clocking switch. A clocking switch also represents a pass transistor, but with its ports labeled as shown in Figure 1b. Clocking switches are used primarily for gating data into data storage components and have an associated clock phase, $\varphi 1$ or $\varphi 2$.

A component type (restoring) logic gate represents the usual NMOS ratio logic element. A component of type logic gate can be decomposed into a component of type pullup and one or more components of type pulldown switch. The component type pullup represents a depletion-mode transistor with its gate tied to its source, as illustrated in Figure 1c. The component type pulldown switch represents an enhancement-mode transistor with its ports labeled as shown in Figure 1d. In the CSG perspective, pullup and pulldown switches can be used only as components of logic gates. The simplest component of type logic gate is an inverter, which consists of a pullup and a pulldown switch connected as shown in

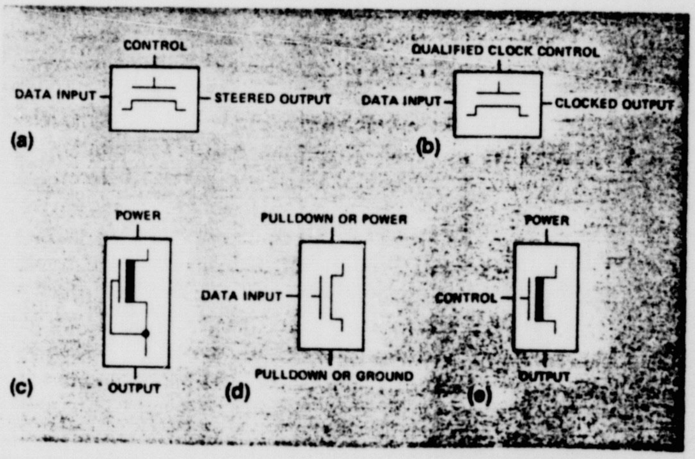


Figure 1. CSG primitive component types: (a) steering switch, (b) clocking switch, (c) pullup, (d) pulldown switch, and (e) controlled pullup.

Figure 2b. Components of type logic-gate performing more complex functions can be constructed from appropriate nets of pulldown switches. For example, a component of type logic-gate performing the function

$$OUT = (NOT((IN1\ AND\ IN2)\ OR\ IN3$$

is shown in Figure 2c.

The component type controlled-gate is a variant of a logic gate. It is composed of a component of type controlled pullup and one or more components of type pulldown switch. The component type controlled pullup represents a depletion-mode transistor with its ports labeled as shown in Figure 1e. An example of a component of type controlled gate is the super-buffer shown in Figure 2d.

The component types defining the CSG perspective are listed in Table 1.

Example 2: A circuit-specific perspective. The Palladio system has been used to create several architectural-level perspectives for investigating the abstract behavior of specific circuits. One such circuit, a MIMD architecture designed by Bruce Delagi of Digital Equipment Corporation, consists of an 8 × 8 grid of nodes, in which processing proceeds by an asynchronous, concurrent message passing between nodes. A node in the grid is composed of a communication chip and a local processor with local

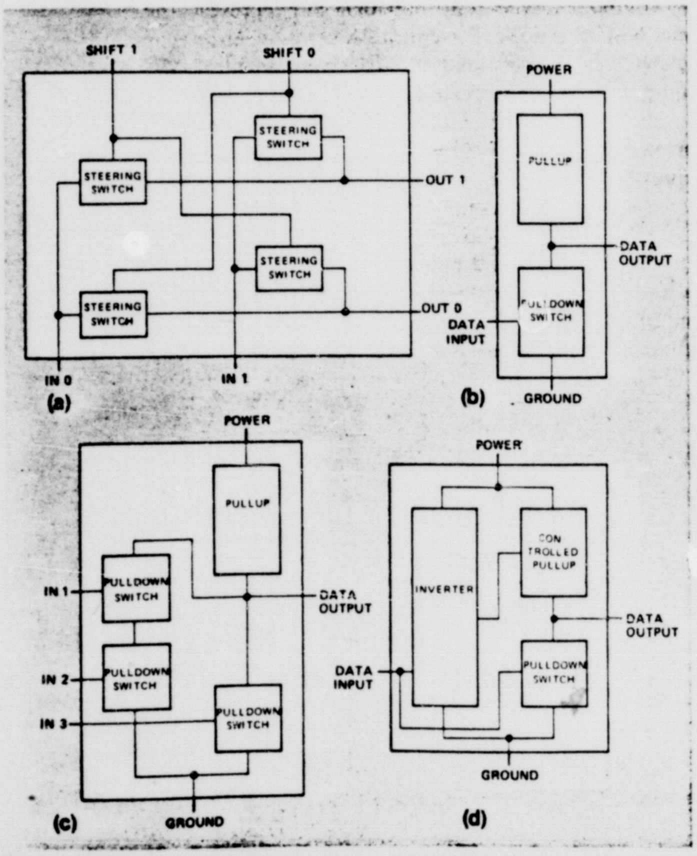


Figure 2. Examples of CSG composite components: (a) 2 × 2 barrel shifter, (b) inverter, (c) logic gate implementing Out = (NOT ((In1∧In2)∨In3)), and (d) inverting super-buffer.

Table 1. Component types defining the clocked switches and gates perspective.

CATEGORY	TYPES
Primitive	Steering switch, clocking switch, pulldown switch, pullup, controlled pullup
Composite	Steering net, clock qualifier, gate, controlled gate, subsystem, circuit
Interconnect	Wire, contact

memory. Each communication chip is doubly connected by byte-width communication paths to its four neighboring communication chips and to its local processor.

The structural perspective created for the circuit has three component types: a communication component type, which has 10 eight-bit-parallel, full-duplex ports and an internal buffer; a processor component with eight-bit-parallel, full-duplex input and output ports; and an eight-bit-parallel wire component. These three-component types suffice to specify the architectural structure of the circuit.

Palladio was used to investigate message congestion in the grid with respect to different protocols for establishing virtual processor-to-processor communication paths and different communication chip buffer lengths. Each communication protocol was described within Palladio by an associated behavior defined for the communication chip. The behavior was expressed from a circuit-specific behavioral perspective based on the concept of message passing. In Palladio, the specification of a component's behavior is transparent and easily modified. We were able to model different communication protocols rapidly and investigate the associated message-passing behavior of the resulting circuit. These investigations produced a refined communication protocol that reduces message congestion.

Composition rules. Through its allowed component types, a structural perspective focuses the concerns of the designer. It can also focus these concerns by limiting the manner in which components are connected. For example, for NMOS layout, the primitive components are regions (e.g., rectangles) of metal, polysilicon, and diffusion. The composition rules are geometric composition rules (e.g., the Lambda rules[13]) that govern size and spacing of the regions. They represent a shallow composition model that is a conservative simplification of a deep model accounting for electrical properties and the fabrication process. The rules ensure that spacing errors will not occur in a correctly fabricated circuit. Such errors could create shorts, opens, or inadvertent transistors, among other problems.

For example, the composition rules of the CSG perspective specify how primitive and composite CSG components can be connected. They represent a shallow model of digital behavior, and they account for voltage level restoration and charge storage. Three of the CSG composition rules are

(1) A control input of a steering switch or a steering net can be connected only to an output of a restoring logic gate. This composition rule forbids the use of a pass transistor's output for driving the gate of another pass tran-

sistor. It reflects a shallow model property, namely that the level of a signal is reduced when it passes through a pass transistor. This reduction cascades when the output of a pass transistor controls another pass transistor. The cascading effect could produce an indeterminate range output signal.

(2) A control input of a clocking switch can be connected only to a basic clock or to an output of a clock qualifier component. This rule enforces the CSG methodology's distinction between pass transistors that clock data and those that steer data.

(3) An output of a clocking switch can be connected only to an input of a restoring logic gate. This rule implies that steering logic cannot be interspersed between clocking switches and logic gates. The rule helps prevent errors in charge sharing that take place when a charge leaks to or from a logic gate storage point (i.e., the input capacitance of the logic gate). Such leaks can result in an indeterminant logic level at the storage point.

Additional CSG composition rules deal with fan-out, series of pass transistors, simple cases of unclocked feedback, and the compatibility of signal levels in interconnection networks. The composition rules for signal-level compatibility prevent power-to-ground shorts and the fan-in of, for example, output of logic gates or pass transistors.

In general, composition rules, such as the Lambda rules,[13] prevent locally detectable errors. More pervasive errors such as illegitimate feedback loops are harder to prevent and often escape detection until some "post-design" verification simulation is applied. Composition rules in Palladio are primarily concerned with avoiding local errors.

Prototype component libraries. Although the component types of a structural perspective are adequate for designing circuits, they are usually not particularly convenient by themselves—experienced designers rarely create circuit specifications directly out of primitives. New circuit design usually makes extensive use of previously designed circuits. For example, a designer may know several alternative implementations for an NMOS state storage device and know how to evaluate these alternatives within the context of a specific circuit. Or new circuit design may consist of modifying some existing circuit design. The use of old designs, particularly of frequently used components (e.g., logic gates and registers) is supported by most design systems. Palladio supports the use of old designs with libraries of prototype components. And associated with each design specification is a library of prototypes which can either be shared by many design specifications or created specifically for a single design.

A prototype library component can be specified from one or more structural or behavioral perspectives. For example, one prototype library in Palladio contains $n \times m$ registers and I/O pads specified from the CSG perspective as well as from more abstract perspectives. From the CSG perspective, registers and pads are composite components that are specified in terms of basic switches and logic gates. From more abstract perspectives, they are primitive; that is, they cannot be decomposed any further.

Figure 3 illustrates the relationships between perspective types, prototype libraries, and components in circuit designs. The prototype library component FOO has two partitionings, one with respect to the CSG perspective and one to a layout perspective. The perspective component types used to specify library components are (virtually) in the library. The components in each partitioning of FOO with respect to a perspective are actually pointers to the appropriate perspective types.

There are three instances of FOO in the circuit design. The circuit components F1 and F2 are instantiated in the circuit from the CSG and layout perspectives, respectively. The partitionings of F1 and F2 in the circuit are specified by the respective partitionings of FOO in the library. The circuit component F3 is an instance of FOO that has no current partitioning—it is specified from a "black-box" perspective. The use of the three different perspectives for the instances of FOO in the circuit might represent a situation in which the designer needs three components, each of whose functionality is the same as that of FOO's. He is satisfied with the library layout for the circuit component F1, wants to lay out the circuit component F2 by hand following the structure given by the CSG perspective of FOO in order to optimize FOO with respect to speed, and is as yet uncertain how to implement the circuit component F3.

The clock switch component appearing in the circuit illustrates that the component types of a perspective are treated as prototype components by any prototype library using the perspective. Thus, instances of perspective types can be used directly in circuit designs.

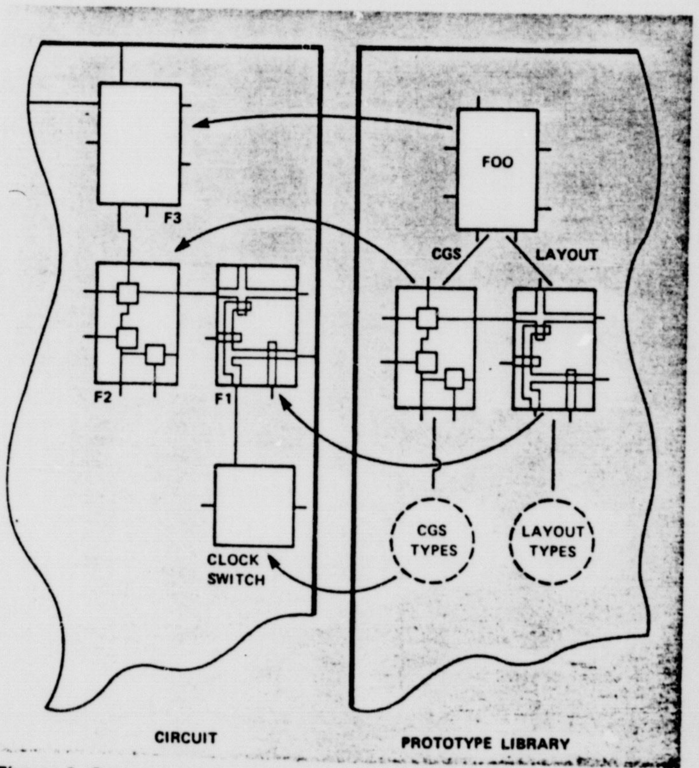


Figure 3. Component prototypes and types.

The design of perspectives. The creation of useful circuit perspectives is a difficult, incremental design process. They represent a significant body of expert knowledge about circuits and circuit design. One of the major purposes of the Palladio environment is to provide mechanisms for easily implementing structural and behavioral perspectives and experimenting with them.

Most of the perspectives currently in use (i.e., design levels) have evolved over a relatively long period, such as the finite state machine, register transfer, gate level, switch level, and symbolic layout. Some, such as the gate level, have their origins in discrete component technologies and may be inappropriate for certain integrated circuit technologies. The derivation of suitable circuit abstractions is an area of active research and development (see Stefik[14]).

In experimenting with different perspectives in Palladio, we have recognized the following four properties as being relevant to the construction of design perspectives:

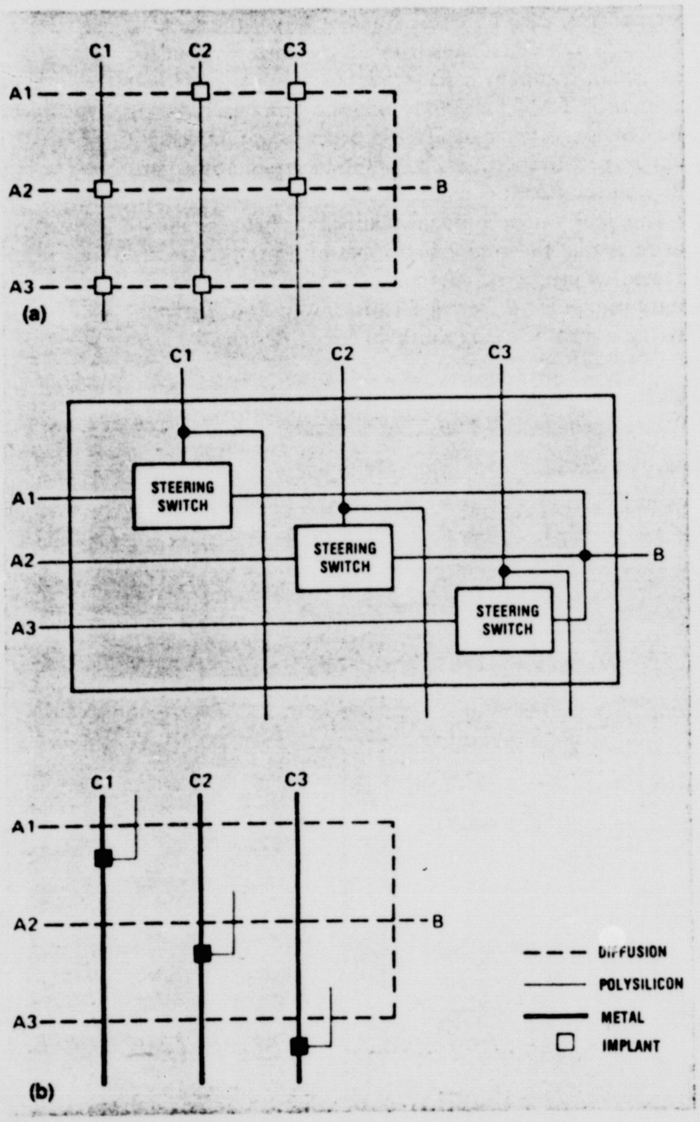


Figure 4. Three-to-one selectors: (a) CSG contravened and (b) CSG allowed.

(1) Structural perspectives complement hierarchical component decomposition. In a hierarchical component decomposition, the degree to which the components can be considered independently—the degree to which the component design can be carried out independently of other components—is directly proportional to the degree of abstractness of the perspective being used to specify the design: The more detailed the specification of the components, the more component interactions must be taken into account, since in any physical circuit, the conceptual components are usually highly interdependent.

Abstract design perspectives allow a designer to deal with less complex specifications for the components and less complex interactions between them. In particular, partitioning is easier with more abstract perspectives, although physical properties such as area and speed can be more difficult to predict from an abstract perspective. Effective use of abstract perspectives and associated component decompositions requires a design methodology that preserves, in the refinement process, the semi-independence of abstractly specified components. Otherwise, the designer could be faced with the impossible task of considering the circuit essentially *in toto* at some later stage in the refinement process. The use of appropriate perspectives helps ensure that the recomposition process is manageable.

(2) A trade-off exists between design optimization and the elimination of certain classes of simple design errors. Design constraints imposed by the perspective (i.e., which circuit constructs are permitted) are balanced against the degree to which the resulting methodology ensures correct circuit specifications. For example, in the CSG perspective, depletion-mode transistors may be used only as pullups, which excludes designs such as the three-to-one selector with sticks diagram as shown in Figure 4a. This selector design gives a very compact layout. However, the resulting circuit is quite sensitive to small variations in the fabrication process. An allowable CSG perspective design for a three-to-one selector along with one possible corresponding sticks diagram is shown in Figure 4b. This second design has a greater area than the first, but the circuit is more robust and less sensitive to irregularities in fabrication.

(3) The closer the components in a circuit's decomposition fit specific areas of an assembled circuit, the easier the process of refinement. The component types of a perspective should be chosen so that the decomposition of a circuit with respect to the perspective agrees reasonably well with the final, physical structure of the circuit. An abstract perspective that permits the partitioning of a circuit into components whose functionality is diffusely spread across any concrete implementation of the circuit (e.g., certain of the "software-like" hardware design languages) is of limited utility in an integrated design environment. Design refinement is very difficult when using such perspectives, but the refinement process is relatively easy to model, and more powerful design refinement tools can be created for perspectives whose component types correspond reasonably directly to their implementations.

(4) Structural and behavioral perspectives are complementary. A behavioral perspective requires a corresponding structural perspective; perspectives of one kind can be coupled with many perspectives of the other kind. Palladio's perspectives emphasize structure or structure-specific behavior; in contrast, some of the hardware design languages tend to emphasize functionality without providing a means for associating structure with function. In Palladio, structural specifications are complemented with behavioral specifications.*

Just as in structural specification, various perspectives can be used in Palladio to specify the behavior of a circuit. Each behavioral perspective provides a designer with a conceptual model and a language for specifying behavior. For example, a behavioral perspective based on a three-valued logic (0, 1, and undefined) views a digital circuit as networks of unidirectional Boolean devices, while a behavioral perspective based on state transition tables views a digital circuit as a finite state machine.

In Palladio, each structural perspective can be associated with one or more behavioral perspectives, and vice versa. For example, with the CSG structural perspective are associated both a three-valued logic perspective and an $n \times m$-valued logic perspective based on Bryant's notion of level-strength pairs[12] that permits bidirectional signals. Conversely, the three-valued logic behavioral perspective is associated with the CSG structural perspective and a clocked register perspective and combinational logic structural level.

Use of multiple perspectives. Using Palladio, we have experimented with a number of different structural and behavioral perspectives and associations between them. Some examples are

- a cell-based, sticks-diagram-with-sized-transistors structure (the SST perspective) with a 3×4-valued level-strength logic function;
- a clocked register and combined logic structure with an associated three-valued logic function;
- a synchronous, finite-state machine structure with a transition-table behavioral perspective;
- a structural perspective whose basic component types include communication nodes and servers with an associated message-sending protocol behavioral perspective for investigating packet-switching networks; and
- a structure whose basic components include task queues, instruction-fetch units, operand-fetch units, registers, cache memories, function units, and instruction counters with associated behavior perspectives at the appropriate levels (e.g., task, instruction, and operand-fetch). This combination has been used to investigate pipelined MIMD architecture.

Since all of these perspectives are implemented in a single system, a component specified from any one perspective can be partitioned using any other perspective. This allows, for example, a digital system specified at an architectural perspective to be incrementally refined through intermediate perspectives to a sticks perspective without ever leaving the Palladio environment—at least in principle. The "Status" section of this article discusses the current limitations of Palladio.

Palladio does not limit the number of perspective-specific partitions associated with a given circuit component. Our initial experiments indicate that multiple behavioral perspectives of a given component are often useful. For example, specifying the behavior of a given component from both a three-valued and a 3×4-valued logic perspective can allow certain economies when simulating a circuit containing that component. However, multiple partitions of a given component into subcomponents with respect to different structural perspectives are rarely used. Multiple partitions—as opposed to the basic refinement process of partitioning a component into subcomponents with respect to one perspective, then partitioning the subcomponent with respect to a different perspective—are of limited utility because it is very difficult to verify that the various component decompositions represent the same circuit. For behavioral perspectives, the interperspective consistency problem is more tractable: For example, there is a direct relationship between three-valued logic and $n \times m$-valued, level-strength logic perspectives.

*We are currently experimenting with a circuit model that spans the entire function-behavior-structure spectrum of digital system specifications.

A partially structured design process

In a fully structured design process (analogous to structured programming), design refinement proceeds uniformly through a sequence of structural perspectives, from the most abstract to the most concrete. First the design is fully specified in terms of the component types of the most abstract perspective. Then each component is specified in terms of the component types of the next most concrete perspective. This process step is repeated until the design is fully specified in terms of the component types of the most concrete perspective. Such a fully structured design process results in a treelike component hierachy, but it is rarely possible to carry out. And even when possible, it often results in a highly suboptimal design.

A fully structured design process has two major problems. First, it requires a complete partitioning of a component into the component types of one structural perspective before considering partitions at a less abstract perspective. The design process is, in part, a continuing trade-off between design objectives as given by the current specification and what can actually be achieved because of limitations imposed, for example, by the device physics or the fabrication technology. Consequently, design is an exploratory process. Designers must decide how much of the overall specification to complete with respect to a given perspective before more concrete specifications for particular components are developed. When they work on more detailed specifications for particular components, they are exploring what can be achieved in the overall design.

A second problem is that a fully structured hardware design process requires a conceptual view of the component hierarchy as treelike. High-level software languages have mechanisms that allow such a viewpoint. For example, a software designer can decompose modules recursively, treating each submodule conceptually as a distinct entity even though two or more submodules may ultimately correspond to the same piece of code. This module decomposition yields a treelike hierarchy, which is allowed because the systems that support the resulting code contain mechanisms for handling shared code (e.g., procedure calls and link loaders). In contrast, structure-sharing in hardware designs is not currently automated.

Figures 5 and 6 illustrate structure sharing. Viewed from a data path perspective, the circuit in Figure 5a consists of a state machine and a data path in which two state-machine outputs are data path inputs. Figure 5b shows the circuit with a clocked register and logic perspective, which treats the two components independently. It permits, in particular, independent editing and simulation of the two components, but does not take advantage of a possible economy through shared structure: The data common to the two components could share a register. A refinement of the circuit that uses the shared register is shown in Figure 5c. This form of structure sharing is viewed as the use of a shared component by the two components.

The circuit in Figure 6 consists of a multiplexer controlled by the external signal *s*, which steers a0 or a1 into a selectively loadable, clocked, one-bit register dependent on the *s* value. The circuit is specified from the CSG perspective in Figure 6a. Its function may be realized more economically by the circuit shown in Figure 6b, in which steering and clocking functions are merged. Steele and Sussman[15] call component decompositions as in the above examples "almost hierarchical."

The forms of structure sharing illustrated above are the only ones admitted by our design paradigm. This allows effective management of the relationships between component (almost) hierarchies and perspectives. Palladio enables the designer to represent shared structure through the use of two distinguished component categories: shared components and merged components. A shared component occurs (virtually) in all components sharing that component. Thus components sharing a component can be independently edited and simulated. In addition, Palladio maintains the relationship between a merged component and the components that it merges. And maintaining such a relationship permits verification (by simulation) that the merged component or its refinements achieve the combined functionality of the components that it merges.

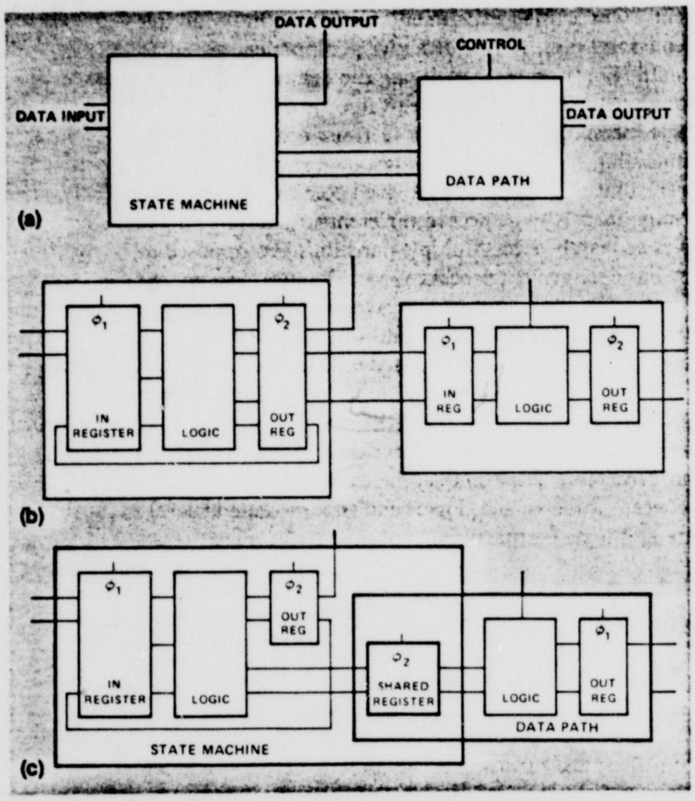


Figure 5. An example of structure sharing: (a) data-path perspective, (b) clocked register and logic perspective, and (c) clocked register and logic perspective with shared register.

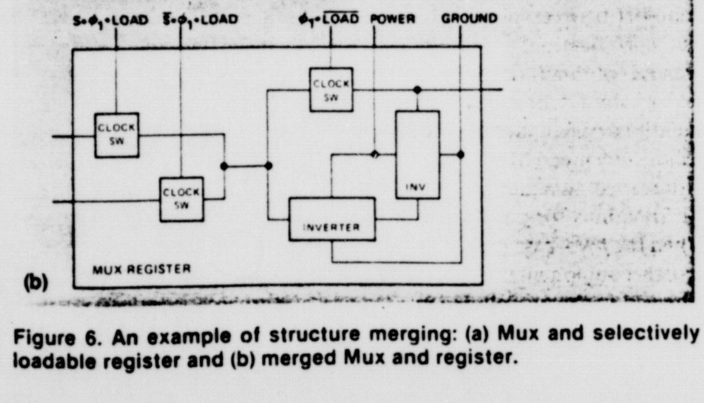


Figure 6. An example of structure merging: (a) Mux and selectively loadable register and (b) merged Mux and register.

Behavioral perspectives

Specifying behavior is an integral part of the circuit design process. The behavior of a digital circuit is the change of its state over time. The state of a circuit consists of the internal state of its components and the values of signals on their ports at a particular time. Palladio models time as discrete and linearly organized contexts.

Behavioral specifications play a critical role in the verification of a design. A refined design specification must meet the design goals and satisfy the constraints imposed by the original specification. Verification is concerned with several aspects of the evolving design: functional behavior, functional performance, design quality (e.g., testability, understandability, robustness), and the possibility of fabrication. A circuit's behavior and performance are usually checked by simulation,* which, in Palladio, means modeling the circuit's structure in the computer, specifying an initial state for the circuit, then using the behavioral specifications of components to infer states in future contexts (not necessarily just the next context) from the current state.

Specification of behavior as rules. Behavior within Palladio is expressed as perspective-specific rules that are triggered by changes in a component's state, and, in turn, change the state of the circuit. A unidirectional pass transistor has three ports: IN, OUT, and CTL. An example of a three-valued logic behavioral rule for a unidirectional pass transistor is:

if *Signal* (*Port CTL*) = *HIGH* at time *t*
then *Signal* (*Port OUT*) = *Signal* (*Port IN*) at time *t* + 1

A different perspective of the pass transistor might use a 3 × 3-valued, level-strength logic.[12] The three following behavioral rules specify the pass transistor's behavior from this perspective:

if *Signal* (*Port CTL*) *Level* = *3*, *Strength* = *s* at time *t*
then *Signal* (*Port OUT*) = *Signal* (*Port IN*) at time *t* + 1

if *Signal* (*Port CTL*) *Level* = *1*, *Strength* = *s1* at time *t*
and *Signal* (*Port OUT*) *Level* = *1*, *Strength* = *s2* at time *t*
then *Signal* (*Port OUT*) *Level* = *1*, *Strength* = *1* at time *t* + 1.

if *Signal* (*Port CTL*) *Level* = *2*, *Strength* = *s* at time *t*
then *Raise error flag*

Motivations for behavioral specification as rules. In an integrated design environment, behavior specification must be in a fairly flexible form, as it must serve many diverse purposes:

(1) The behavioral specification is part of the overall design specification and must be in a form the designer can enter and later comprehend. The rule format has the advantage of being fairly transparent and well structured.

(2) The behavioral specification must be usable by a simulator; furthermore, the system should be able to simulate connected components with behavioral specifications in different perspectives. The rule format can be used to express any kind of behavior that can be expressed as computation (as it permits embedded calls to Lisp functions). In particular, the rule format can accommodate behavior that transcends traditional logic modes of digital design. For example, the rule syntax permits Boolean logic control to be integrated with high-level function units:

if *Signal* (*Port CTL*) = *HIGH* at time *t*
then *Signal* (*Port OUT*) = *Signal* (*Port IN1*) times *Signal* (*Port IN2*)

(3) A component's behavioral specification must be compatible with its structural specification. It can be compared, by simulation, with the behavioral specifications of its interconnected components to verify component decomposition. For example, in Figure 7, the HALF-REG component could be verified by simulations using its indicated behavioral specification and the behavior of its components.

(4) The behavioral specification may serve as a constraint or as input for other programs, such as an automatic refinement program. For example, behavior described from a Boolean logic perspective could be input to a PLA generator to produce a layout perspective design. Rules express behavior in a form convenient for input to other programs.

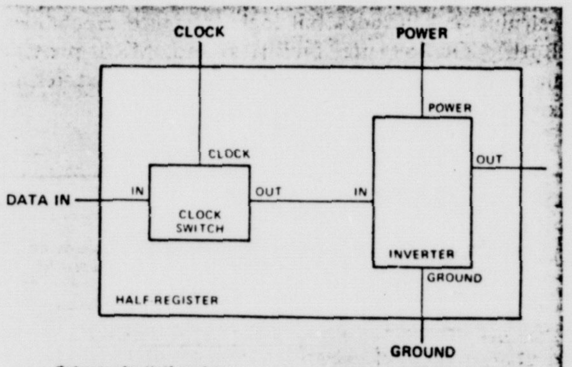
(5) It should be possible to specify behavior for "dummy components" that are not implemented but are used for generating or monitoring signals during a simulation. Such dummy components are analogous to PRINT statements used to debug a software program but are afterwards removed from the program.

(6) The behavioral specification should help explain a particular simulation result. It is easy to produce primitive explanations from records of rule activations.[3]

Programming language procedures could likewise be used for behavioral specification, as they too are very flexible. However, procedures are generally less comprehensible than rules, can be difficult to use as input to other programs, and admit no simple explanation facility.

Palladio's simulator

Palladio's simulator is based on MARS[17] (Multiple Abstraction Rule-based Simulator), a general-purpose, event-driven simulator whose versatility derives, in part, from the logic reasoning system in which it is implemented, the Multilevel Reasoning System.[5] A logic reasoning system contains, as data, a set of assertions and a collection triggered by the presence of assertions and capable of producing new assertions. It uses inference rules to control how new assertions are added.



Behavior for Half-register:

IF Signal(Port Clock) = HIGH at time *T*
THEN Signal(Port Out) = INVERT Signal(Port In) at time *T* + *3*.

Behavior for Clock Switch:

IF Signal(Port Clock) = HIGH at time *T*
THEN Signal(Port Out) = Signal(Port In) at time *T* + *1*.

Behavior for Inverter:

IF Signal(Port IN) = *s* at time *T*
THEN Signal(Port Out) = INVERT Signal(Port In) at time *T* + *2*.

Figure 7. Verification of behavior.

*There is some current work on using formal methods for behavioral verification of circuits; see, for example, Barrow.[16]

MARS expresses the state of a circuit as a set of assertions and directly maps the behavioral rule formalism described earlier into the MRS rule formalism. Repeated use of the inference rule *modus ponens* (that is, if *A* and *A*→*B*, then *B*) produces new circuit states by causing MRS to cycle through the state-representing assertions, applying all applicable behavioral rules to each assertion and adding new assertions to the end of the current set of assertions.

This very general simulation framework allows the hybrid simulation of high-level, sparsely detailed functional blocks and low-level, highly detailed gates and switches. By simulating at highly detailed perspectives only when it is necessary to verify the design from that perspective, the component hierarchy can be exploited for simulation performance gains. In a typical hierarchical simulation, most components are simulated from their high-level behavioral perspective—either out of necessity (as their structure has not been fleshed out) or because their low-level behavior has already been verified.

In Palladio, a simulator run can be dynamically displayed on a color graphics screen. In a logic simulation, for example, attribute values can be denoted by distinct colors. This dynamic display produces a "movie" of the circuit's behavior. Also a formatted text file of a simulation can also be saved for later analysis.

Implementation

Palladio is implemented with object-oriented, data-oriented, rule-based, and logic-reasoning programming paradigms. The object-oriented, data-oriented, and rule-based facilities are provided largely by the LOOPS (Lisp Object-Oriented Programming System) programming environment.[4] LOOPS is implemented in Interlisp-D,[18] a programming environment that augments Interlisp[8] with interactive bitmap graphics, multiple processes, and networking capabilities. Interlisp-D runs on the Xerox D-series workstations.

The MRS logic language system is based on predicate calculus and includes full logic inference capabilities. Both LOOPS rule facilities and MRS provide mechanisms for developing, integrating, and testing knowledge-based expert system design aids.

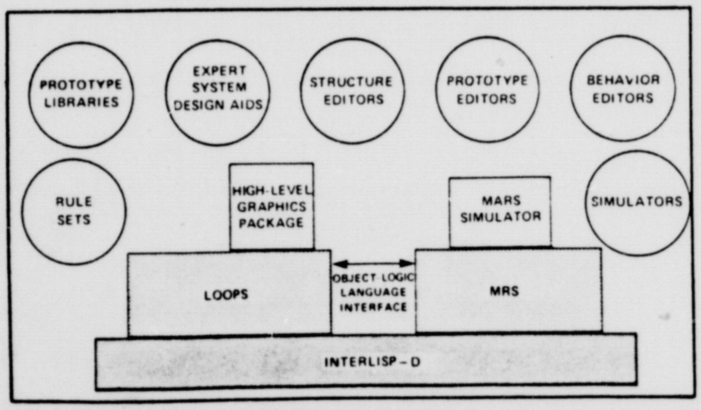


Figure 8. Palladio system architecture.

The overall Palladio software architecture is illustrated in Figure 8. The circled entities in the figure are groups of objects, while the boxed entities are supporting systems.

Object-oriented programming. Palladio is implemented mainly in an object-oriented programming paradigm whose basic entities are objects and messages (see Smalltalk[19]). All major system components—structure editors, behavior editors, and simulators—are represented as objects, representational packages that include a data structure and a set of methods (i.e., procedures) for operating on that structure. Palladio circuit entities—circuit, components, ports, wires, etc.—are also treated as objects.

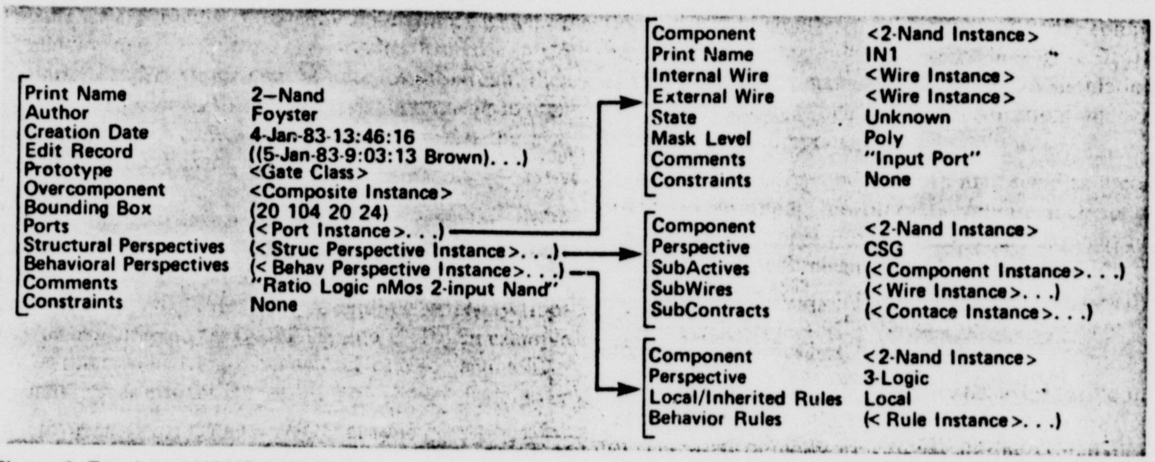
Large classes of objects often share identical data structures, differing only in the values of those data structures. For instance, all wires have two ends and an associated signal type and signal strength. Object-oriented languages exploit this class property that defines a basic data structure and its associated methods. A class serves as a template for creating instances, which are objects sharing the data structure of their class. In Palladio, a component type is defined by a class object, and any prototype instance in a circuit specification is an instance object of the prototype class object.

The data structure of a LOOPS object is a frame composed of attribute-value pairs. The frame of a two-input NAND gate instance specified from the CSG perspective is shown in Figure 9. The value of an attribute can be any LOOPS or Lisp datatype (atom, list, array, object, active value), and the values of some of the attributes are pointers to other objects. This is denoted in the figure by values of the form <*x*>, which stands for "a pointer to an object of type *x*."

A class describes its instances by specifying the names and default values of attributes. Instances of a class are created with attributes as described in its class. In its attribute values, an instance contains the information that distinguishes it from other instances of its class. The frame for the class "wire" is shown in Figure 10, and an instance of "wire" is shown in Figure 11.

Object-oriented programming provides a powerful, flexible solution to the problem of representing generic actions because every type of entity provides its own definition for a generic action, such as displaying itself on a screen. A message sent to an object invokes the response pattern associated with that message in the definition of the object's class. For example, a Palladio editor or simulator is invoked for a particular circuit by sending an ACTIVATE message to the editor or simulator; to display a component in a screen window, a DISPLAY message is sent to the component; and to add an instance of a prototype component to a circuit, a MAKE-INSTANCE message is sent to the type's class. The message-passing technique is a natural means for creating software modularity. And the message sender need know only that the recipient can respond to the message. It does not need to know how the recipient will respond. The recipient of the message knows the appropriate response method and how to invoke it.

Palladio's class inheritance network. Object-oriented programming uses the class/instance distinction to ex-



Print Name	2—Nand
Author	Foyster
Creation Date	4-Jan-83-13:46:16
Edit Record	((5-Jan-83-9:03:13 Brown). . .)
Prototype	<Gate Class>
Overcomponent	<Composite Instance>
Bounding Box	(20 104 20 24)
Ports	(<Port Instance>. . .)
Structural Perspectives	(<Struc Perspective Instance>. . .)
Behavioral Perspectives	(<Behav Perspective Instance>. . .)
Comments	"Ratio Logic nMos 2-input Nand"
Constraints	None

Component	<2-Nand Instance>
Print Name	IN1
Internal Wire	<Wire Instance>
External Wire	<Wire Instance>
State	Unknown
Mask Level	Poly
Comments	"Input Port"
Constraints	None

Component	<2-Nand Instance>
Perspective	CSG
SubActives	(<Component Instance>. . .)
SubWires	(<Wire Instance>. . .)
SubContracts	(<Contace Instance>. . .)

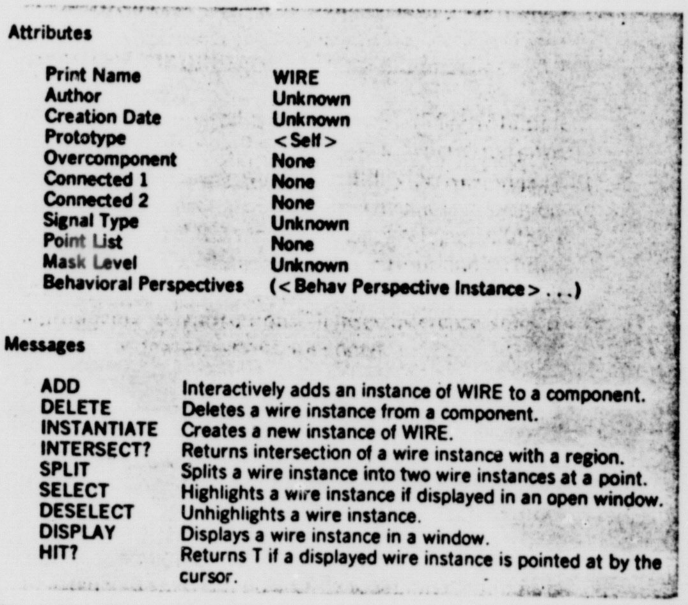
Component	<2-Nand Instance>
Perspective	3-Logic
Local/Inherited Rules	Local
Behavior Rules	(< Rule Instance>. . .)

Figure 9. Two-input NAND gate frame.

ploit the fact that objects often share identical object data structures and methods. It also takes advantage of similar, but not identical, data structures and methods. In LOOPS, classes are organized into an inheritance network; a subclass inherits the attribute descriptions of, and the messages understood, by its parent classes. A subclass can also have noninherited attributes and messages with their associated methods. Moreover, the default values of inherited attributes and the associated methods of inherited messages can differ from those of the parents.

Part of the class inheritance network for Palladio's circuit objects is shown in Figure 12. As indicated in the figure, classes fall into three categories. The kernel classes (e.g., Composite) are part of a special set of classes that define the basic Palladio environment. The generic classes (e.g., State Composite) are used in the definition of one or more circuit perspectives. Kernel and generic classes are used for definitions only; no instances are ever created. The third category, prototype classes, contains prototype circuit components (register and NAND gate). Each prototype class is a subclass of a prototype, generic, or kernel class that differs from its parent only in its default attribute values. The class inheritance mechanism is used to create new prototype components or new perspectives. For instance, a three-input NAND gate prototype class could be created as as subclass of either the gate or NAND class.

Palladio provides interactive graphics editors for defining new prototype components. The definition of a new perspective, however, can involve the creation of new generic classes without attributes and methods inherited from their kernel parents. Currently, such classes are created with the LOOPS object editor, and the process requires familiarity with the underlying object representations and Interlisp.



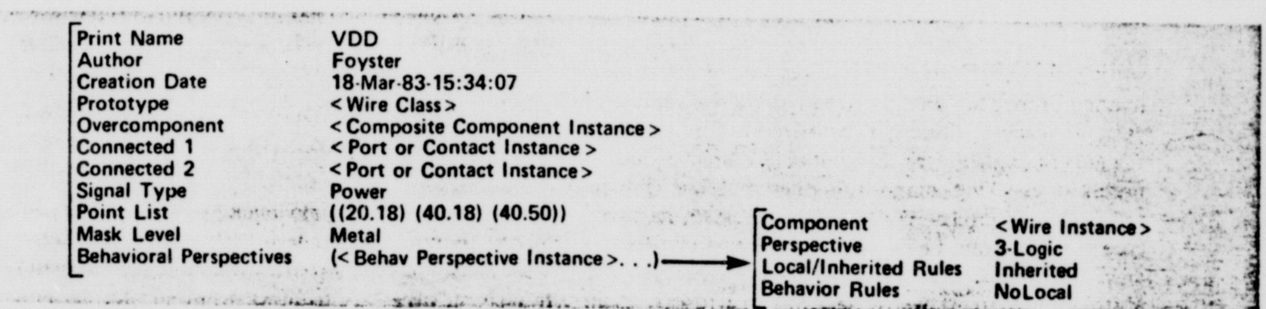
Attributes

Print Name	WIRE
Author	Unknown
Creation Date	Unknown
Prototype	<Self>
Overcomponent	None
Connected 1	None
Connected 2	None
Signal Type	Unknown
Point List	None
Mask Level	Unknown
Behavioral Perspectives	(<Behav Perspective Instance> . . .)

Messages

ADD	Interactively adds an instance of WIRE to a component.
DELETE	Deletes a wire instance from a component.
INSTANTIATE	Creates a new instance of WIRE.
INTERSECT?	Returns intersection of a wire instance with a region.
SPLIT	Splits a wire instance into two wire instances at a point.
SELECT	Highlights a wire instance if displayed in an open window.
DESELECT	Unhighlights a wire instance.
DISPLAY	Displays a wire instance in a window.
HIT?	Returns T if a displayed wire instance is pointed at by the cursor.

Figure 10. Wire class object.



Print Name	VDD
Author	Foyster
Creation Date	18-Mar-83-15:34:07
Prototype	<Wire Class>
Overcomponent	<Composite Component Instance>
Connected 1	<Port or Contact Instance>
Connected 2	<Port or Contact Instance>
Signal Type	Power
Point List	((20.18) (40.18) (40.50))
Mask Level	Metal
Behavioral Perspectives	(<Behav Perspective Instance>. . .)

Component	<Wire Instance>
Perspective	3-Logic
Local/Inherited Rules	Inherited
Behavior Rules	NoLocal

Figure 11. Wire instance object.

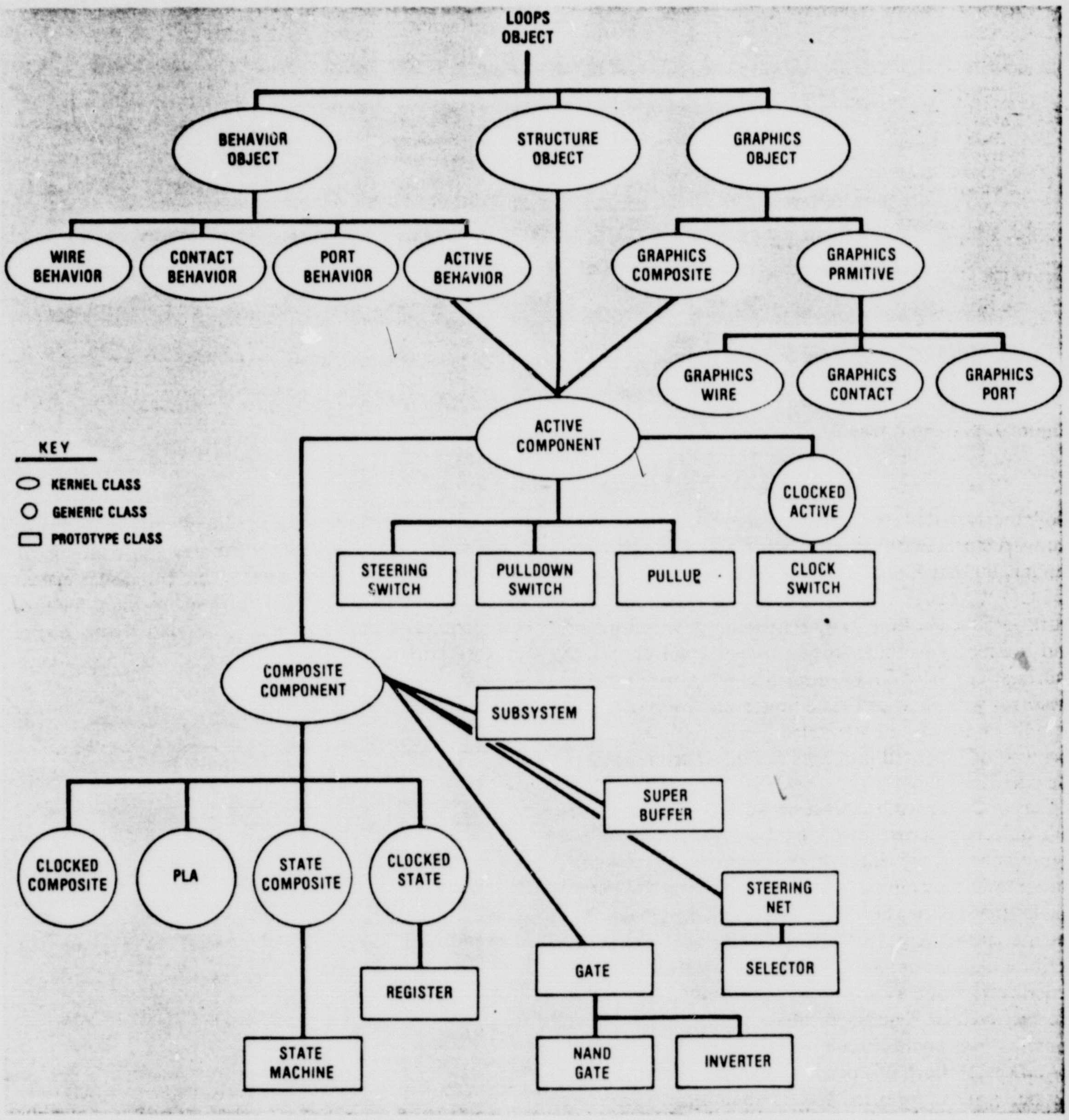


Figure 12. Partial class inheritance network.

Data-oriented programming. In data-oriented programming, reading or writing on the value of a particular attribute of an object causes attribute-specific side effects. LOOPS permits data-oriented programming through a notation that allows the programmer to distinguish passive and active attribute values. Reading or writing on an active value produces side effects by activating a procedure associated with the active value. For example, each CSG perspective wire has an associated signal type (e.g., Power, Ground, $\varphi 1$, Passed), which is useful for checking composition rules or assigning interconnect mask levels. When a wire is created, its signal type cannot always be determined immediately; for example, the signal type of a wire connecting ports of two abstract subsystems whose internal structures remain unspecified is indeterminate. Eventually, the design becomes sufficiently detailed that the signal type of the wire can be inferred from its connections. An active value is used as a data demon that (conceptually) monitors the signal type attribute on each wire. Whenever that value is set, the procedure propagates the signal type throughout the net and verifies that the signal type is consistent within the net.

Design tools as expert systems

We are developing knowledge-based design aids based on the perspective framework provided by Palladio. They are small expert systems that perform some of the refinement necessary to move from an abstract circuit specification to more concrete circuit specifications. By providing a uniform framework of multiple perspectives, Palladio simplifies the implementation of such expert systems.

The expertise needed by such design aids can take various forms, such as an algorithm to implement registers as gates or heuristic knowledge expressed symbolically. We are currently experimenting with circuit design aids as rule-based expert systems.[20]

An example of a design aid. Given a design specified from the CSG perspective, one of the layout refinements is to assign mask levels to the interconnect (wires and contacts) between the components. A design tool that performs this task permits a designer to see the consequences of circuit refinement quickly and allows him to avoid unnecessary layout errors.

The goal is to produce a cell-based sticks diagram for the circuit. The strategy we have used begins at the most detailed level of the circuit's component hierarchy and progresses to the most abstract, assigning mask levels (e.g., metal, polysilicon, or diffusion) to interconnect along the way. Thus, after a port of a component C_1 is attached to a wire with a particular mask level, the mask-level assigner tries to maintain the established mask-level assignment when considering a component C_2, which contains C_1.

The mask-level assignment strategy considers such qualitative factors as minimizing power and delay and introducing as few vias (vertical channels connecting wires with different mask levels) as possible. It avoids unintended connections between intersecting wires and inadvertant transistors, such as those made by intersecting polysilicon and diffusion.

To keep the problem feasible, a constraint was introduced: The planar topology of the interconnect must remain as given in the CSG perspective. Components and wires must remain in the same relative positions after mask-level assignment.

Many expert systems distinguish base-level actions from control-level actions: Base-level actions modify the representation of the problem and its solution, while control-level actions determine which base-level actions to take. Control-level actions can be represented as rules of the form

if *situation-predicate* then *action*

where the predicate tests for the existence of a particular situation before performing its action. The mask-level assigner is implemented with Lisp procedures for the base-level actions and LOOPS rules to control Lisp procedures. There are two base-level actions: (1) assignment of a mask level to a wire and (2) introduction of vias for changing the mask level along a wire.

The overall strategy followed by the mask level assigner is

(1) identify all wire intersections;
(2) order the intersections according to a set of rules; and
(3) for each intersection, apply rules that
 (a) determine whether mask levels need to be assigned or reassigned for the two intersecting wires,
 (b) generate costs associated with different mask levels for each wire,
 (c) generate costs for the various types of vias, and
 (d) invoke base-level procedures to produce least-cost assignments for the wires using the derived costs.

This stategy focuses on one intersection at a time, even with wires that intersect many other wires. When producing an assignment for a wire by focusing on a single intersection, the mask-level assigner can inadvertently introduce a short or transistor at some other intersection. This means that step (3) in the above strategy must be repeated until all wires have been assigned a mask level, and there are no unintentional shorts or transistors.

The control-rule sets take into account factors such as the current mask level (if any) of a wire, the estimated length of a wire, the signal type it carries (e.g., power or clock), and the total number of intersections along a wire. Figure 13 gives examples of control rules.

Because Palladio allows access to any part of a circuit's structure, the rule sets can make use of whatever global information is necessary to achieve high performance within a highly focused (localized) control strategy. For example, the ordering of intersections uses a set of rules to determine the relative importance of each intersection. This rule set requires global information such as the types of components connected by a wire and the connection networks of intersecting wires. Easy and quick access to circuit information is a central factor in the high performance and quick implementation of the mask-level assigner.

Implementation of the mask-level assigner. LOOPS provides rule-oriented programming in which a rule set can be associated with an object. Individual rules in a set can test the attribute values of the object to conditionally execute procedures, set attribute values, or send messages. When a set is invoked, the rules are tested and the conditional actions are executed.

```
Determine if mask levels need to be reassigned
        IF Wire1:MaskLevel=POLY
          AND
          Wire2:MaskLevel=DIFF
        THEN ReassignMaskLevels

Order treatment of wires
        IF Wire1:Signal=Power
          AND
          Wire2:Signal=PHI1
        THEN Assign Wire1 before Wire2

Specify costs for different layers for a wire
        IF Wire:Endport1=POLY
        THEN Cost(POLY)-LOW

Invoke base level actions
        IF MaskLevelNeedsAssignment
          AND
          MaskCostsDefinedForWire
        THEN AssignLowestCostToWire
```

Figure 13. Examples of control rules for the wire mask assigner.

The "wire assigner" is an object created to carry out the mask-level assignment. It responds to the following messages:

- The ACTIVATE message takes a circuit as a parameter and initializes the assignment process.
- FINDINTERSECTIONS finds all intersections between wires in the specified circuit, creates a new intersection object for each, and stores the objects in a list.
- REORDERINTERSECTIONS sorts the intersection object list according to the importance of the intersections, as determined by a set of rules.
- MAKEASSIGNMENT invokes the appropriate base-level actions to assign a mask level (including vias) to a wire.

Intersection objects respond to the MASKCONFLICT? and RESOLVECONFLICT messages. RESOLVE CONFLICT is sent to an intersection object if a conflict exists, that is, if MASKCONFLICT? responds affirmatively. The assignment process terminates when all the wires are assigned mask levels and there are no further mask-level conflicts.

The MASKCONFLICT? and RESOLVECONFLICT methods for an intersection object are implemented as rule sets. They determine the priority for two conflicting wires, as well as the cost associated with different mask levels for each wire. To generate the minimal cost assignment for a given wire, a "wirecosts" object is created containing attributes for determining costs of various mask levels. When cost values have been established, GENERATEMINASSIGNMENT is sent to the wirecosts object resulting in a search through possible assignments (where introducing vias is considered legitimate). When the wirecosts object finds the lowest cost assignment, it sends MAKEASSIGNMENT to the wire assigner, and the assignment is made. The process can be dynamically displayed on the color screen.

Infinite loops in the wire assignment process could occur because a prior wire assignment can be undone by another assignment of the same wire when considering a different intersection. We have eliminated this possibility by introducing a cost for each intersection, which is the sum of the costs of previous assignments to the two wires. By increasing the intersection cost each time a conflict is resolved, the assignment process is forced to terminate.

Benefits of an expert systems approach. After its basic strategy was outlined, the mask-level assignment expert system was implemented in two days. The initial system contained few rules. The assignments were error free but of poor overall quality. For example, unnecessary vias were introduced, and power and ground wires were run in polysilicon or diffusion for no good reason. Adding rules that accounted for the types of signals running in a wire and that ordered wire assignments more efficiently greatly improved the resulting sticks diagrams.

The current system produces mask-level assignments for large-scale circuits comparable to those produced by human designers. The mask-level assigner demonstrates one of the premises of expert system construction: Performance improves as knowledge is added. And specifying the control knowledge as rules has made the system easy to understand and modify.

Status

The basic Palladio framework has been operational for about a year. The current system provides

(1) Interactive graphics editors that treat components as rectangular boxes with attached ports. Wires, connect ports, and components can be partitioned into subcomponents. Components that are added to an evolving design are selected from standard or designer-created libraries of prototype components. Prototype component editors can be invoked from within circuit design editors, allowing a designer to easily augment a prototype component library during the circuit design process.

(2) A behavioral rule editor that gives syntactic support for entering and modifying behavioral specifications of both prototype components and circuit components.

(3) An event-driven simulator that uses the behavioral and structural specifications of a circuit to simulate it. The simulator can perform hierarchical simulation (use either the specified behavior of a component or the behavior induced by its subcomponents and their interconnections) and mixed-perspective simulation (e.g., simulation of a circuit in which some of the components have behavior specified from a three-valued logic perspective and some specified from a 3×4-valued logic level.

(4) A frame-based mechanism for assigning multiple perspectives to components. The mechanism also allows those limited forms of nonhierarchical component decomposition that we have found useful in Palladio.

(5) A protocol for creating new structural and behavioral perspectives based on Palladio's object-oriented paradigm.

(6) Mechanisms for implementing rule-based, expert-system design aids.

Our initial (and current) implementation of Palladio was a research effort. Our interest was in investigating circuit design environments. With Palladio, we designed several circuits using perspectives ranging from architectural through cell-based, sticks diagram levels. Even this limited experience has confirmed the value of many of Palladio's underlying concepts, particularly the hierarchical use of multiple perspectives, distinct structural and behavioral perspectives, behavioral specifications based on a rule format, a behavioral simulator applicable to all levels of behavioral specification, and design aids implemented as rule-based expert systems. In its current implementation, however, Palladio is difficult for anyone other than its builders to use. It is slow, requiring about eight hours on a Xerox 1100 computer to run a 1000-event simulation of the multiprocessor circuit described earlier.

During the implementation of Palladio, we were often uncertain about what system capabilities would prove to be useful. Whenever we were faced with a flexibility-versus-efficiency trade-off, we opted for flexibility. However, we have paid a price for flexibility. Running on a Xerox 1100, we can deal adequately only with circuit designs consisting of tens of high-level components and, at most, hundreds of low-level components.

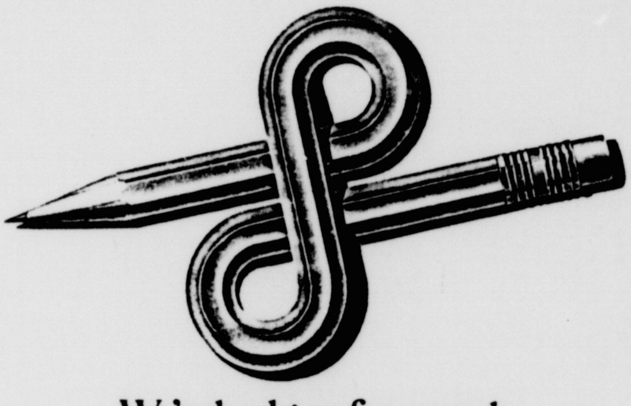
The current implementation of Palladio is overly general. Some of the system's capabilities, such as the underlying representational mechanisms for circuit structure and behavior, have limited use. We are currently modifying Palladio, observing more realistic flexibility-versus-efficiency trade-offs. This modification should greatly improve performance. But even with major improvements and with the powerful Xerox 1132 computer, we estimate that Palladio will remain limited to circuits with, at most, tens of thousands of low-level components. We conjecture that flexible, fully integrated design environments for custom, VLSI-scale circuits will require computers more powerful than those that are currently available.

Palladio is an early attempt to explore the stuff on which circuit design environments are built. This exploratory design environment recognizes the need to integrate diverse design tools and design languages; perspectives are an attempt at creating the flexible framework required to support experiments with such tools and languages. In our Palladio investigations, we have acknowledged that the construction of the "perfect set" of design tools and languages is a never-ending process that must keep pace with the ever-expanding boundaries of circuit technology and of computer-aided design. It requires representation of the tools and languages in an easily modifiable, expandable form.

**Table 2.
Programming paradigms and their design environment applications.**

PARADIGM	APPLICATION
Object oriented	Structural specification
Logical language	Behavioral specification and simulation
Rule based	Incrementally constructed expert design aids
Data oriented	Constraint propagation

Table 2 summarizes how we used programming paradigms for building different elements of the design environment. Multiple paradigms proved useful for representing diverse kinds of tools and languages explicitly and for making their modification and extension as straightforward and rapid as possible. The table indicates only a few preliminary correspondences; finding best fits between programming paradigms and design environment applications is a novel, but important area for research. Palladio is an exploratory design environment that contains an exploratory programming environment to facilitate experiments with varying elements of an integrated design. ■



We're looking for people who don't think straight.

If you're the kind of person who is able to look around the corner instead of straight ahead, consider a career with Linkabit.

Linkabit is a leader in the design, development and manufacture of satellite and terrestrial communications equipment specializing in local communication networks, encryption, forward error correction and high speed modulation.

And the reason we are is that our people are very creative, free-thinking individuals who look beyond the obvious to find innovative solutions to the demanding problems of this industry.

To help keep new ideas flowing, we've made sure that all career paths at Linkabit are flexible. Our engineers, for instance, are assigned to projects depending on their interests and abilities. As one assignment is completed, new opportunities are made available in a variety of areas.

Utilizing the most advanced equipment available, we work on a number of diverse and complex projects which include MILSTAR terminals, video scrambling equipment, domestic satellite systems, modems, codecs, advanced processors and fault-tolerant systems.

We are currently seeking skilled individuals who possess applicable experience in the following areas:

- **Software Design & Development**
- **Digital Circuit Design**
- **VLSI**
- **Communications System Design**
- **Software Support/Tools Development**
- **Thermal & Package Engineering**
- **Microprocessor Based Systems**
- **Components Engineering**
- **Project Engineering/Program Management**

Positions are available in San Diego, Washington, D.C. and Boston. Send your resume to: Dennis Vincent, M/A-COM LINKABIT, 3033 Science Park Road, San Diego, CA 92121.

M/A-COM

M/A-COM LINKABIT, INC.
Equal Opportunity/
Affirmative Action Employer

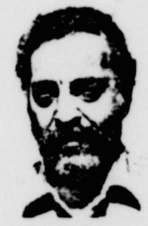
Acknowledgments

Thanks to Phillip Gerring for designing and implementing the object-oriented graphics package used in Palladio and to Jerry Yan for his contributions to the interconnect mask-level assigner. Thanks also to our colleagues Allen Bell, Daniel G. Bobrow, Lynn Conway, and Mark Stefik at the Xerox Palo Alto Research Center and Harry Barrow at the Fairchild Laboratory for Artificial Intelligence Research, all of whom worked with us in Palladio's initial stages. Several of the ideas presented in this article are drawn from the results of this early collaboration.

Our research was supported in part by the Defense Advanced Research Projects Agency under contract N0039-83-C-0136. Xerox PARC provided the Xerox 1100 workstation on which we implemented Palladio.

References

1. J. Werner, "Sorting Out the CAE Workstations," *VLSI Design*, Vol. 4, No. 2, Mar./Apr. 1983, pp. 46-55.
2. D. S. Nau, "Expert Computer Systems," *Computer*, Vol. 16, No. 2, Feb. 1983, pp. 63-85.
3. *Building Expert Systems*, F. Hayes-Roth, D. Waterman, and D. Lenat, eds., Addison-Wesley, Reading, Mass., 1983.
4. D. G. Bobrow and M. Stefik, "The LOOPS Manual," memo KB-VLSI-81-13, Xerox Palo Alto Research Center, Palo Alto, Calif., Aug. 1981, rev. Aug. 1982.
5. M. Genesereth, "MRS: A Metalevel Representation System," working paper HPP-83-28, Stanford University, Heuristic Programming Project, June 1983.
6. A. Barr and E. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. 1, William Kaufman, Los Altos, Calif., 1982, chapt. 3.
7. C. U. Smith and J. A. Dallen, "A Comparison of Design Strategies for Software and VLSI," *Proc. Compcon 83*, Feb. 1983, pp. 263-268.
8. W. Teitleman, *INTERLISP Reference Manual*, Xerox Corporation, Palo Alto Research Center, Palo Alto, Calif. 1978.
9. B. A. Shiel, "Power Tools for Programmers," *Datamation*, Vol. 29, No. 2, Feb. 1983, pp. 131-144.
10. H. Simon, *The Sciences of the Artificial*, 2nd ed., MIT Press, Cambridge, Mass., p. 148.
11. M. Stefik et al., "The Partitioning of Concerns in Digital Systems Design," in *Proc. Conf. Advanced Research in VLSI*, P. Penfield, ed., Artech House, Dedham, Mass., Jan. 1982.
12. R. E. Bryant, *A Switch-Level Model and Simulator for MOS Digital Systems*, tech. report 5065, Caltech, Pasadena, Calif. 1983.
13. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.
14. M. Stefik and L. Conway, "Towards the Principled Engineering of Knowledge," *AI Magazine*, Vol. 3, No. 3, summer 1983, pp. 4-16.
15. G. J. Sussman and G. L. Steele, Jr., "CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions" *Artificial Intelligence*, Vol. 14, No. 1, Aug. 1980, pp. 1-39.
16. H. G. Barrow, "Proving the Correctness of Digital Hardware Designs," *Proc. AAAI*, William Kaufman, Los Altos, Calif., 1983, pp. 17-21.
17. N. Singh, *MARS: A Multiple Abstractions Rule-Based Simulator*, tech. report 17, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, Calif., 1983.
18. *Cognitive and Instructional Sciences Series CIS-5*, B. A. Sheil and L. M. Masinter, eds., Xerox Palo Alto Research Center, Palo Alto, Calif., Sept. 1980, rev. Jan. 1983.
19. A. Goldberg, "Introducing the Smalltalk-80 System" *Byte*, Vol. 6, No. 8, Aug. 1981, pp. 36-48.
20. B. G. Buchanan and R. O. Duda, "Principles of Rule-Based Expert Systems," in *Advances in Computers*, M. Yovits, ed., Vol. 22, Academic Press, New York, 1983, pp. 163-216.



Harold Brown is a senior research associate in the Department of Computer Science at Stanford University. He is a member of Stanford's Heuristic Programming Project, an interdisciplinary laboratory for research in artificial intelligence and applications of knowledge-based systems. His current interests include knowledge-based systems for computer aided design and for understanding remotely sensed signals and computer architectures for supporting such systems.

Brown received an MS in mathematics from the University of Notre Dame in 1963 and a PhD in mathematics from the Ohio State University in 1966.

Christopher Tong is completing a PhD in computer science, with an emphasis on artificial intelligence, at the Heuristic Programming Project, Department of Computer Science, Stanford University. Tong also consults for the Knowledge Systems Area at Xerox PARC. His research interests include the development of a comprehensive model of design and the creation of computer environments that implement such design models. His previous positions include work at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, where he developed several innovative algorithms for analyzing circuit stability.

Tong is a member of the AAAI, Phi Beta Kappa, and Upsilon Pi Epsilon. He received a BA in computing science summa cum laude from Columbia University, New York, in 1978.

Gordon Foyster is a member of the research staff of the Heuristic Programming Project within the Department of Computer Science at Stanford University. He is involved in research in expert systems, especially in circuit design and the implementation of frameworks supporting such expert systems. Previously, he worked as a systems engineer developing real-time avionics instruments at Ferranti in Scotland. Foyster received a BS in mathematics from Wollongong University, Australia, in 1979 and an MS in computer science from Stanford in 1983.

Questions about this article can be directed to Harold Brown, Dept. of Computer Science, Stanford University, Stanford, CA 94305.