June 1983
Also numbered: HPP-83-31

Report No. STAN-CS-83-984

1

Palladio:
An Exploratory Environment for Circuit Design

by

Harold Brown, Christopher Tong
and Gordon Foyster

Department of Computer Science

Stanford University
Stanford, CA 94305

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

Heuristic Programming Project
HPP Report HPP-83-31

June 1983

Palladio:
An Exploratory Environment for Circuit Design

HAROLD BROWN
CHRISTOPHER TONG
GORDON FOYSTER

HEURISTIC PROGRAMMING PROJECT
DEPARTMENT OF COMPUTER SCIENCE
STANFORD UNIVERSITY

To appear in *IEEE Computer*, December 1983

This research is funded by the Defense Advanced Research Projects Agency
under Contract N0039-83-C-0136. (ONR)

Palladio:

An Exploratory Environment for Circuit Design

Harold Brown, Christopher Tong, and Gordon Foyster

Stanford University

Introduction

*Palladio** is a circuit design environment for experimenting with methodologies and knowledge-based, expert-system design aids. Palladio's framework is based on several premises about circuit design: (a) circuit design is a process of incremental refinement; (b) it is an exploratory process in which design specifications and design goals co-evolve; and (c) most importantly, circuit designers need an *integrated design environment* that makes available compatible design tools ranging from simulators to layout generators, 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.

**Andrea Palladio* (1518-1580) was the Italian architect who developed the methodology of proportion and formal architectural style that has become known as *classical* architecture. In a sense he was the first *knowledge engineer* of design principles, and his influential published works are still in print four hundred years after his death.

Organization of the Paper

In the introductory section we discuss the concept of an integrated design environment. In the next three sections we describe the basic conceptual framework underlying Palladio: In the section **The Circuit Design Process**, we discuss Palladio's view of the design process as incremental refinement; in the section **Design Perspectives**, we introduce the central concept of Palladio--design specification using structural and behavioral design perspectives; and in the section **A Partially Structured Design Process** we describe how Palladio permits certain deviations from a fully hierarchical component decomposition process. The next section, **Behavior Specification**, describes Palladio's use of a rule format for behavioral specification, and the system's simulator is described in the section **Palladio's Simulator.** The section **Implementation** discusses how Palladio is implemented using multiple programming paradigms. An example of an expert-system design aid implemented using Palladio is given in the section **Design Tools as Expert Systems**. The current status of the Palladio system and our future plans are described in the concluding sections **Status** and **Conclusions.**

Integrated Design Environments

There is a growing trend toward creating integrated design environments, as opposed to isolated design aids. During the past year, several commercial computer-aided engineering (CAE) workstations for assisting the circuit designer have emerged.[1] These workstations provide multiple-level circuit specification entry systems (e.g., functional specifications and schematics) and analysis aids (e.g., logic simulators and timing verifiers). 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. The Palladio environment integrates both design tools and design specification languages and is an attempt to formalize the type of conceptual framework required by such an integration. That such integration is only now taking place reflects the diversity and complexity of the kinds of expert design knowledge that need to be integrated.

An integrated design environment reduces or eliminates many needs artificially induced by lack of integration. The need for certain design tools is a by-product 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 some long chain of decisions, only to take back some of them when their negative consequences finally become apparent. For example, a designer may implement a multiplexer component using random gate logic and then later retract that implementation when it becomes apparent that a PLA implementation would be more area efficient. Such a lag between the time of a decision and the time when its consequences can be analyzed is, in part, inherent in the exploratory nature of design; however, some of this need for backtracking can be eliminated given a common framework in which one can describe known laws, heuristics, and trade-offs connecting high-level decisions (e.g., architectural decisions) with low-level consequences (e.g., area of a layout).

Palladio differs from other integrated design environments by providing the means for *constructing*, *testing*, and incrementally *modifying* or *augmenting* design tools and languages. For the circuit designer, Palladio supports the construction of new specification languages particular to the design task at hand and augmentation of the the system's expert knowledge to reflect current design constraints and goals. For the design environment builder, Palladio provides several programming paradigms: rule based (expert systems oriented),* 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] The differing features of these paradigms (described in the section **Implementation**) make each useful for constructing different elements of a design environment.

*See, for example, Nau[2] or Hayes-Roth[3] for an introduction to expert systems.

Perhaps the most significant property these programming paradigms share is that each makes possible the *explicit* representation of specific kinds of design knowledge. A representation is explicit if it can be treated and manipulated as data by a program. Much of artificial intelligence research in the last ten years can be viewed as contributing to a classification of different kinds of knowledge and exploiting explicit knowledge represcntions.[6] Designing an integrated circuit requires a significant body of *expert knowledge*, taking a wide variety of forms and often technology specific or even circuit-type specific. The expert knowledge used by a design environment should be available to a designer in an understandable and easily accessible form. Much valuable expert knowledge can take the form of a well-integrated collection of task-specific design aids. One of the primary uses of Palladio is as a vehicle for acquiring and recording the knowledge used by such design aids.

Elements of the Palladio Environment

Palladio provides a test bed 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 to interactively create and refine circuit design specifications. Perspectives can include composition rules that constrain how circuit components may be combined in that perspective to form more complex components.

Palladio 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."

Palladio has 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

The design of a circuit can be viewed as the transformation of an initial specification of the circuit into a final specification that adequately details how the circuit is to be realized physically. The initial specification, which is usually imprecise and incomplete, focuses primarily on the circuit's *functionality.* The final specification emphasizes the *structure* of the circuit (e.g., 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 is physically realizable is an *incremental* process in which abstract specifications of the circuit's structure and behavior are gradually *refined* into more detailed specifications. During the refinement process, the evolving design specifications must be tested and evaluated against the goals and constraints of the design, that is, the design must be verified. A circuit design environment should provide tools to assist the designer in both design specification and design verification throughout the refinement process.

Circuit design in many ways resembles the design of complex software systems (see, e.g., Smith[7]). The design of circuits is more analogous to exploratory programming than it is to structured programming. 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 the design of the program's specifications can evolve together, as the programmer has available a well-integrated set of powerful tools to assist in the development process (e.g., file management, interactive graphics, and debugging tools).[9] An exploratory circuit design environment, like an exploratory programming environment, must provide an integrated set of tools to assist the designer.

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 (and, thus, 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 (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.[10]

This top-down technique of *hierarchical partitioning* is used universally by designers to simplify the design process. The process of partitioning a component into constituent components to reduce the complexity of the design process is 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. Using appropriate design specification levels helps ensure that the recomposition process is tractable.

The design paradigm supported by Palladio is incremental refinement of design specifications, interspersed with validation of the specifications by simulation. The basic hierarchical step in structural refinement is the partitioning of a component 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. Other permissible refinement steps are adding detail to the structural specification of a component (e.g., clock phase to a register or mask level to a wire) and specifying behavior with respect to some behavioral description level. 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 using *design perspectives.* A design perspective provides both a *conceptual model* for viewing the structure or the behavior of a circuit that emphasizes certain of the circuit's features while suppressing others and 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, and specified using state transition tables or a register-transfer language. A perspective in Palladio is either structural or behavioral. This explicit decoupling of behavioral perspectives from structural perspectives allows a modularity not admitted by most circuit design languages; one or more behavioral perspectives can be associated with a structural perspective and vice versa. This modularity permits, for example, a component to be specified from a very abstract behavioral perspective at an early stage in the design process and from a more concrete behavioral perspective at a later stage in the process. Moreover, a circuit designer can use Palladio to construct structural and behavioral perspectives specifically tailored for a particular circuit. (Some examples of such circuit-specific perspectives are given 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, a circuit can be partitioned into only registers, combinational logic blocks and interconnect. Some of the component types allowed by a perspective are *primitive* with respect to the perspective, that is, they cannot be partitioned from the perspective; while other allowed component types are *composite* with respect to the perspective, that is, 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* with respect to a perspective if all of the components at the lowest level of the circuit's component hierarchy are primitive with respect to the perspective.

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 the 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. Using appropriate perspectives helps ensure that the recomposition process is tractable.

Examples of Structural Perspectives

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

Example 1: CSG Perspective. One of the more concrete experimental perspectives (i.e., its specifications are close to being physically realizable) we have implemented is the *Clocked Switches and Gates* (CSG) perspective. The CSG perspective views a circuit as networks of steering logic, clocking logic, and restoring logic gates, and it is specialized for nMOS circuits that use a two phase, nonoverlapping clocking scheme. The perspective is concerned primarily with the digital behavior of a circuit, 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 (e.g., Bryant[12]).

The primitive component type for steering logic is a *steering switch.* A steering switch represents a pass transistor with its ports labeled as shown in Figure 1a.

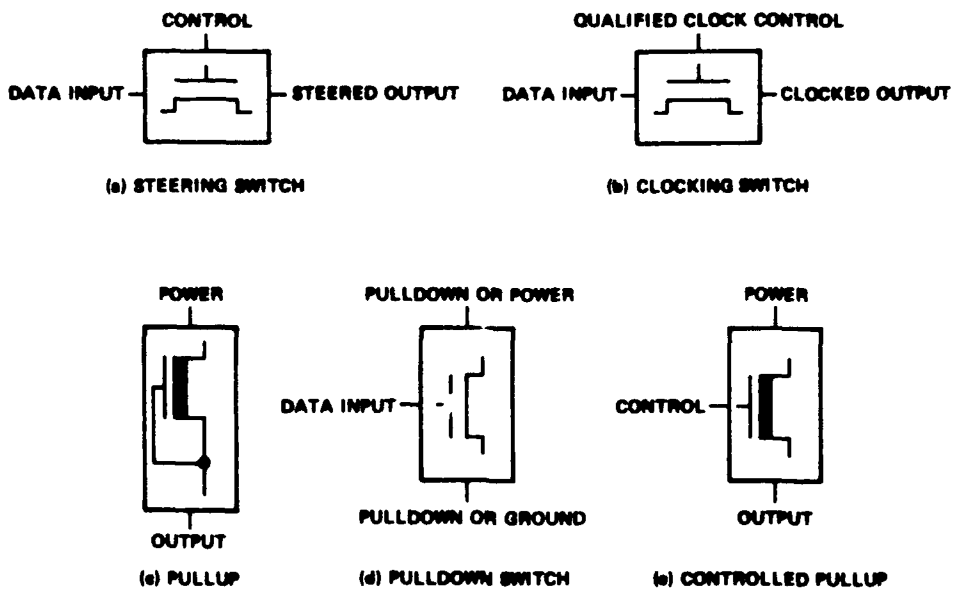


Figure 1 CSG primitive component types.

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, $\varphi1$ or $\varphi2$.

A *(restoring) logic gate* component type represents the usual nMOS ratio logic element. A component of type logic gate can be decomposed into a component of type pullup and

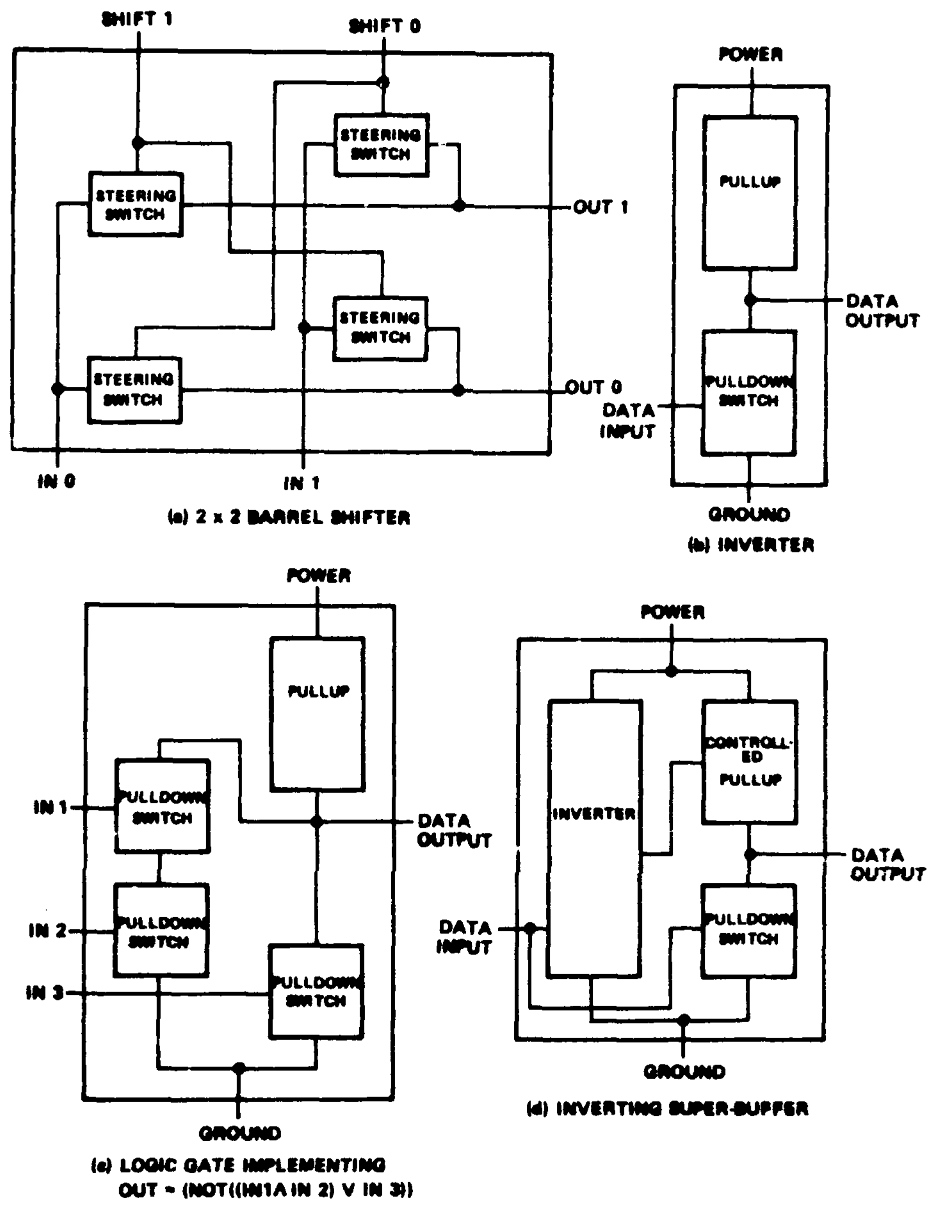


Figure 2. Examples of CSG composite components.

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

$$\text{Out} = (\mathit{NOT}((\text{In1}\ \mathit{AND}\ \text{In2})\ \mathit{OR}\ \text{In3}))$$

is shown in Figure 2c.

A component of 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 1c. 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.

Table 1.

CSG Component Types.

Primitive Components

Steering Switch
Clocking Switch
Pulldown Switch
Pullup
Controlled Pullup

Composite Components

Steering Net
Clock Qualifier
Gate
Controlled Gate
Subsystem
Circuit

Interconnect Components

Wire
Contact

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 8X8 grid of nodes where processing proceeds by concurrent message passing between nodes. A node in the grid is composed of a communication chip and a local processor with local memory. Each communication chip is doubly connected to its four neighboring communication chips and to its local processor. All communication paths in the circuit are byte-wide.

The structural perspective created for the circuit has three component types: a communication component which has ten 8-bit-parallel, full-duplex ports and an internal buffer; a processor component with full-duplex input and output ports, both 8-bit-parallel; and an 8-bit-parallel wire component. These three component types suffice to specify the architectural structure of the circuit.

The purpose of the investigation was to evaluate message congestion in the grid under different protocols for establishing virtual processor-to-processor communication paths and using different length buffers in the communication chip. Each communication protocol was implemented in Palladio by defining an associated behavior for the communication chip using 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 (see the section **Behavioral Perspectives**). Thus, we were able to rapidly model different communication protocols and investigate the associated message passing behavior of the resulting circuit using Palladio's simulator. These investigations resulted in a refined communication protocol which exhibits much less message congestion than the originally proposed protocols.

Composition Rules

A structural perspective focuses the concerns of the designer by its allowed component types. It can also focus these concerns by limiting with *composition rules* 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 for layout are geometric rules (e.g., the Lambda rules[13]) governing the sizes and spacings of the regions. These geometric composition rules provide a shallow model of composition that is a conservative simplification of a deep model of composition accounting for electrical properties and the fabrication process. Following the rules ensures 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 are based on a shallow model of digital behavior, and they account for voltage level restoration and charge storage. Three of the composition rules for the CSG perspective are:

Rule 1. *A control input of a steering switch or a steering net can only be connected 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 transistor. This prohibition reflects a shallow model property, namely, that the level of a signal is reduced when it passes through a pass transistor and that this reduction cascades when the output of a pass transistor is used to control another pass transistor. The cascading effect could result in an output signal in the indeterminate range.

Rule 2. *A control input of a clocking switch can only be connected to a basic clock or to an output of a clock qualifier component.*

This rule enforces the CSG methodology's distinction between those pass transistors that clock data and those that steer data.

Rule 3. *An output of a clocking switch can only be connected 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 charge sharing errors. Charge sharing takes place when a charge is allowed to leak to or from a logic gate storage point (i.e., the input capacitance of the logic gate) and can result in an indeterminant logic level at the storage point.

Additional CSG composition rules deal with fanout, series of pass transistors, simple cases of unclocked feedback, and the compatibility of signal levels in interconnection networks. The composition rules for signal level compatability prevent power-to-ground shorts and the fan-in of, for example, the outputs of a logic gate and a pass transistor.

In general, locally detectable errors are those most readily constrained away using composition rules; the Lambda rules for layout design are an example.[13] Errors of a more global nature (e.g., illegitimate feedback loops) are harder to prevent and often escape detection until some "post-design" verification technique like 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 structural specification, they are usually not particularly convenient; experienced designers rarely create circuit specifications directly out of primitives. The design of a new circuit usually makes extensive use of previously designed circuits. For example, a designer may know several alternative implementations for a state storage device in nMOS circuits and know how to evaluate these alternatives within the context of a specific circuit, or designing a new circuit may consist of modifying some existing circuit design. The use of existing designs, particularly of frequently used components (e.g., logic gates and registers), is supported by most design systems. Palladio supports the use of existing designs with libraries of prototype components. Associated with each design specification is a library of prototypes used in the specification. This prototype library can either be shared by many designs or created specifically for the design of a single circuit.

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 which 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 with respect to a layout perspective. The perspective types for each perspective used in the library 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.

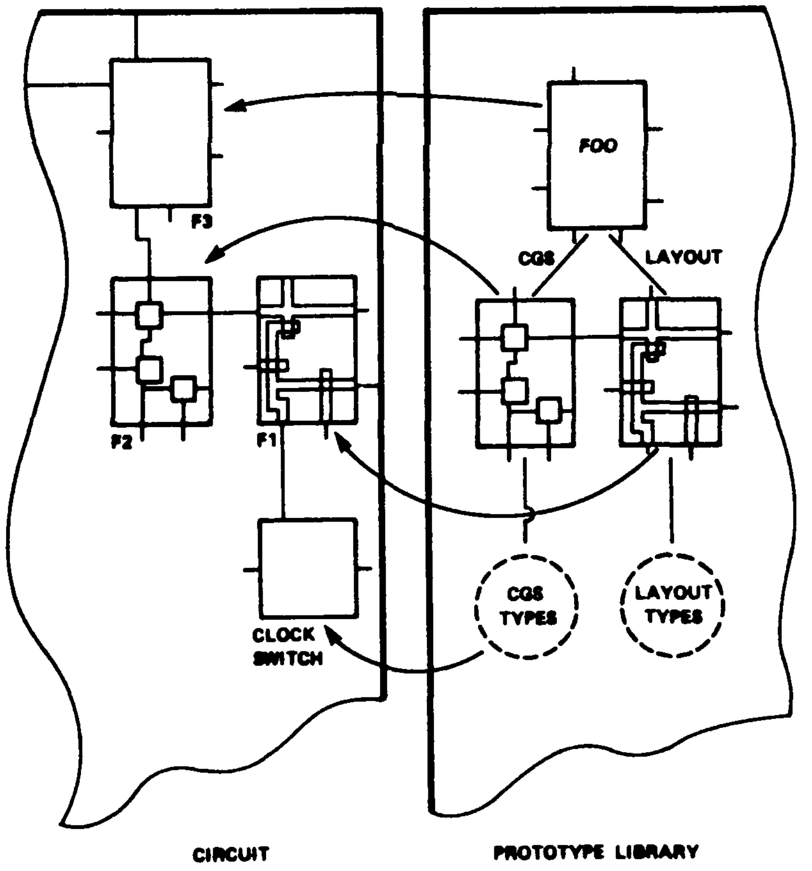


Figure 3. Component prototypes and types.

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. That is, the partitionings of *F1* and *F2* in the circuit are as specified by the respective partitionings of *FOO* in the librbary. The circuit component *F3* is an instance of *FOO* which has no current partitioning (i.e., 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 where the designer needs three components each of whose functionality is the same as that of *FOO's* and a) is satisfied with the library layout for the circuit component *F1*, b) wants to hand layout the circuit component *F2* following the structure given by the CSG perspective of *FOO* in order to optimize *FOO* with respect to speed, and c) 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.

The Design of Perspectives

The creation of useful circuit perspectives is a difficult, incremental design process. Most of the perspectives currently in use (i.e., design levels) have evolved over a relatively long period (e.g., finite state machine, register transfer, gate level, switch level and, symbolic layout). Some of these levels (e.g., 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, e.g., "*Towards the Principled Engineering of Knowledge*"[14]). A circuit perspective represents 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 and experimenting with structural and behavioral perspectives.

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

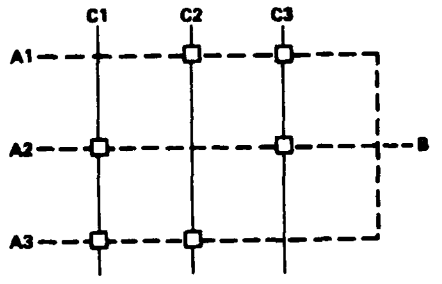
Property 1. *The use of structural perspectives and hierarchical component decomposition is complementary.*

The use of structural perspectives complements the component decomposition process. In a hierarchical component decomposition, the degree to which the components can be considered independently, that is, the degree to which the design of a component can be carried out *independent* of the design of the 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. This reflects the fact that in any physical circuit resulting from the design, the (conceptual) components are usually highly interdependent.

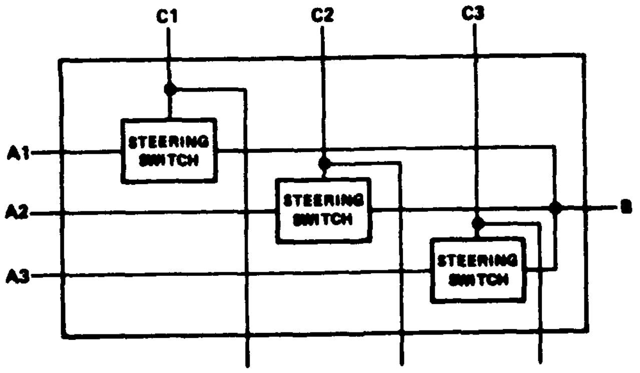
The use of abstract perspectives throughout the design process provides leverage by allowing a designer to deal with less complex specifications for the components and less complex interactions between them. In particular, the partitioning process is easier at the more abstract perspectives (although physical properties such as area and speed can be more difficult to predict from an abstract perspective). The effective use of abstract perspectives and associated component decompositions requires following a design methodology that ensures, in the refinement process, the preservation of the semi-independence of abstractly specified components. Otherwise, the designer could be faced with the impossible task of having to consider 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 tractable.

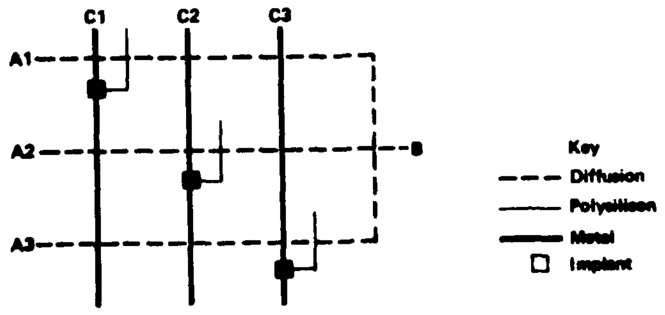
Property 2. *A trade-off exists between the optimality of a design and the guaranteed absence of certain classes of simple errors in that design.*

Construction of a design perspective can involve trading off flexibility and optimality against the guaranteeable absence of certain simple classes of design errors; the constraints placed on the designer by the perspective (i.e., which circuit constructs are permitted) are balanced against the degree to which the resulting methodology ensures the correctness of circuit specifications with respect to the perspective's concerns. For example, in the CSG perspective, depletion mode transistors may only be used as pullups. This constraint excludes designs such as the 3-to-1 selector with sticks diagram as shown in Figure 4a. This selector design results in a very compact layout; however, the resulting circuit is quite sensitive to small variations in the fabrication process.



(a) CSG CONTRAVENED SELECTOR





(b) CSG ALLOWED SELECTOR

Figure 4. 3-to-1 selectors.

An allowable CSG perspective design for a 3-to-1 selector along with one possible corresponding sticks diagram is shown in Figure 4b. This second design leads to a layout with greater area than the first design, but, the resulting circuit is more robust and less sensitive to fabrication tolerances.

Property 3. *The greater the correspondence between the components in a circuit's decomposition and specific areas of a physical realization of the circuit, the easier the process of refinement.*

The component types of a perspective should be chosen so that resulting component decompositions agree reasonably well with the final physical structure of the circuit (i.e., there is a well-defined, but not necessarily one-to-one, correspondence between perspective-specific components and particular areas of the physical 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. 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.

Property 4. *Structural and behavioral perspectives complement each other: A behavioral perspective depends on the existence of an associated 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 function with structure. In Palladio, structural specifications are complemented with behavioral specifications.*

*We are currently experimenting with a single conceptual model of circuits that spans the function-behavior-structure spectrum of digital system 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 3-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, the CSG structural perspective has associated with it both a 3-valued logic perspective and an $n \times m$-valued logic perspective (based on Bryant's notion of level-strength pairs[12]) that admits bidirectional signals. Conversely, the 3-valued logic behavioral perspective is associated with both the CSG structural perspective and a clocked register 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. Examples include:

A cell-based, sticks-diagram-with-sized-transistors structural perspective (the *SST* perspective) with an associated 3×4-valued level-strength logic behavioral perspective.

A clocked registers and combinational logic structural perspective with an associated 3-valued logic behavioral perspective.

A synchronous finite state machine structural perspective with an associated state 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.

A structural perspective whose basic component types include task queues, instruction fetch units, operand fetch units, registers, cache memories, function units, and instruction counters with associated behavior at the appropriate levels (e.g., task, instruction, and operand-fetch). This perspective has been used to investigate a pipelined style of 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; see the section Status for the current limitations of Palladio).

Palladio does not constrain the number of perspective-specific partitions associated with a given circuit component. Our initial experiments seem to indicate that multiple behavioral perspectives of a given component are often useful. For example, specifying the behavior of a given component both from a 3-valued logic perspective and from a 3 $\times$ 4-valued logic perspective can allow certain economics when simulating a circuit containing that component. However, parallel partitionings of a given component into subcomponents with respect to different structural perspectives are rarely used (as opposed to partitioning a component into subcomponents with respect to one perspective and then partitioning the subcomponents with respect to another perspective). For example, directly partitioning a given component both into a collection of finite state machines and into a collection of logic gates is of limited utility because it is very difficult to verify that the two component decompositions are consistent (i.e., represent the same circuit). For behavioral perspectives, the interperspective consistency problem is more tractable; for example, there is a direct relationship between a 3-valued logic and an $n \times m$ level-strength logic.

A Partially Structured Design Process

In the Palladio environment, the specification of a circuit component is the totality of its existing perspectives. At any point in the design process, a component's specification can consist of complete or partial specifications from one or more perspectives. The refinement of a component's specification proceeds by an iterative sequence of steps, each of which alters the values of certain attributes of the component with respect to some perspective.

In a strictly structured design process (analogous to structured programming), design refinement proceeds uniformly through a hierarchy of structural perspectives, from the most abstract to the most concrete, partitioning components hierarchically along the way. That is, the design is first fully specified from the most abstract perspective; the components at this perspective are then, in turn, partitioned into components, either all specified from the original perspective or all specified at the next more detailed perspective. This process results in a treelike component hierarchy. In general, such a fully structured hardware design process is not feasible; even in the cases where it is possible, it often results in highly suboptimal designs.

A fully structured design process has two major problems. The first is that it requires a complete partitioning of a component into the primitives 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, that is, it is an exploratory process. Designers must decide how much of the overall specification at a given perspective to complete before more concrete specifications for particular components are developed. When designers work on more detailed specifications for particular components, they are exploring what can be achieved for the overall design.

A second problem with a fully structured hardware design process is that it requires *conceptually* viewing the component hierarchy as tree-like. 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. Such a simple conceptual hierarchy 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.

The following two examples 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 of the outputs of the state machine are the inputs to the data path.

A refinement of the circuit from a clocked register and logic perspective is illustrated in Figure 5b. This refinement treats the two components independently and permits, in particular, independent editing and simulation of the two components. However, the refinement does not take advantage of a possible economy gained 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.

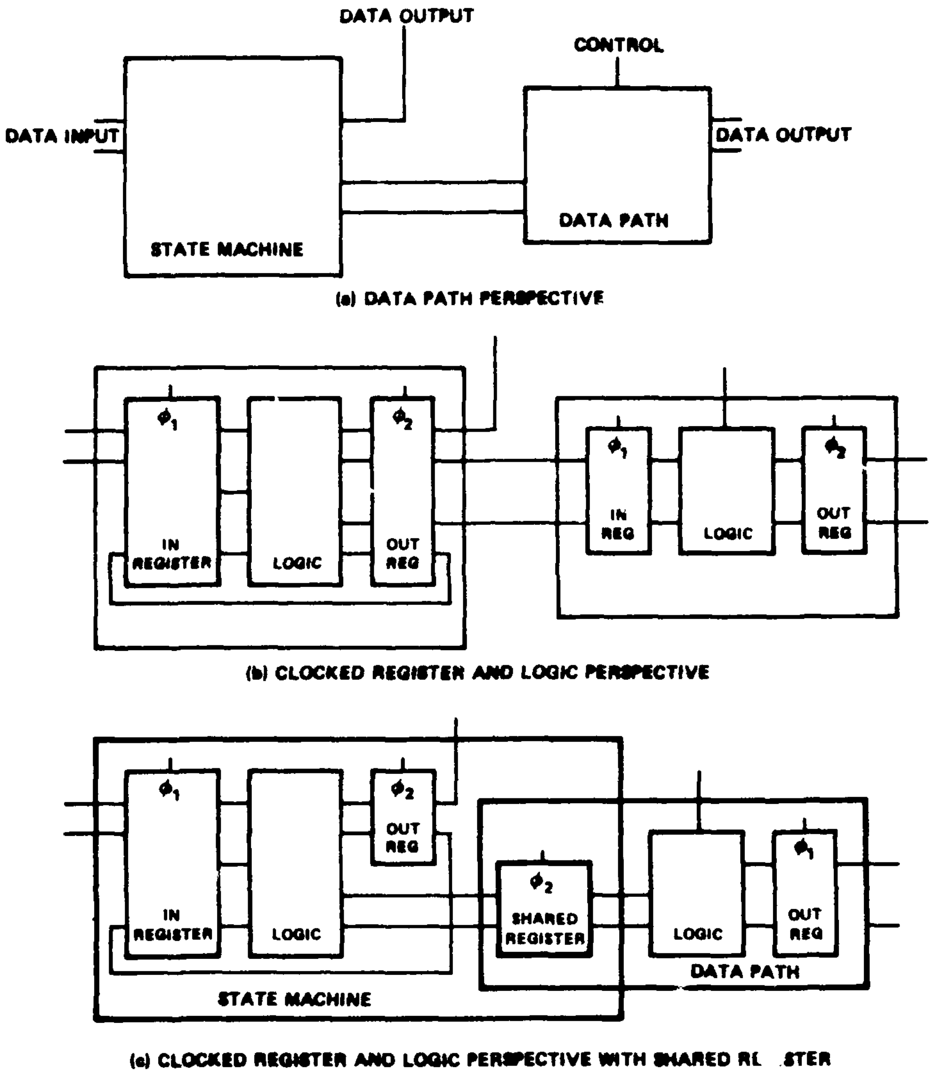


Figure 5. An example of structure sharing.

A second example of structure sharing is illustrated in Figure 6. The circuit consists of a multiplexer controlled by the external signal *s* which steers *a0* or *a1* into a selectively loadable, clocked, 1-bit register dependent on the value of *s*. The circuit is specified from the CSG perspective in Figure 6a.

The circuit's function may be realized more economically by the circuit shown in Figure 6b, which merges the steering and clocking functions of the two components. This form of structure-sharing is viewed as the *merging* of the two components. Steele and Sussman[15] call component decompositions as in the above examples *almost-hierarchical*.

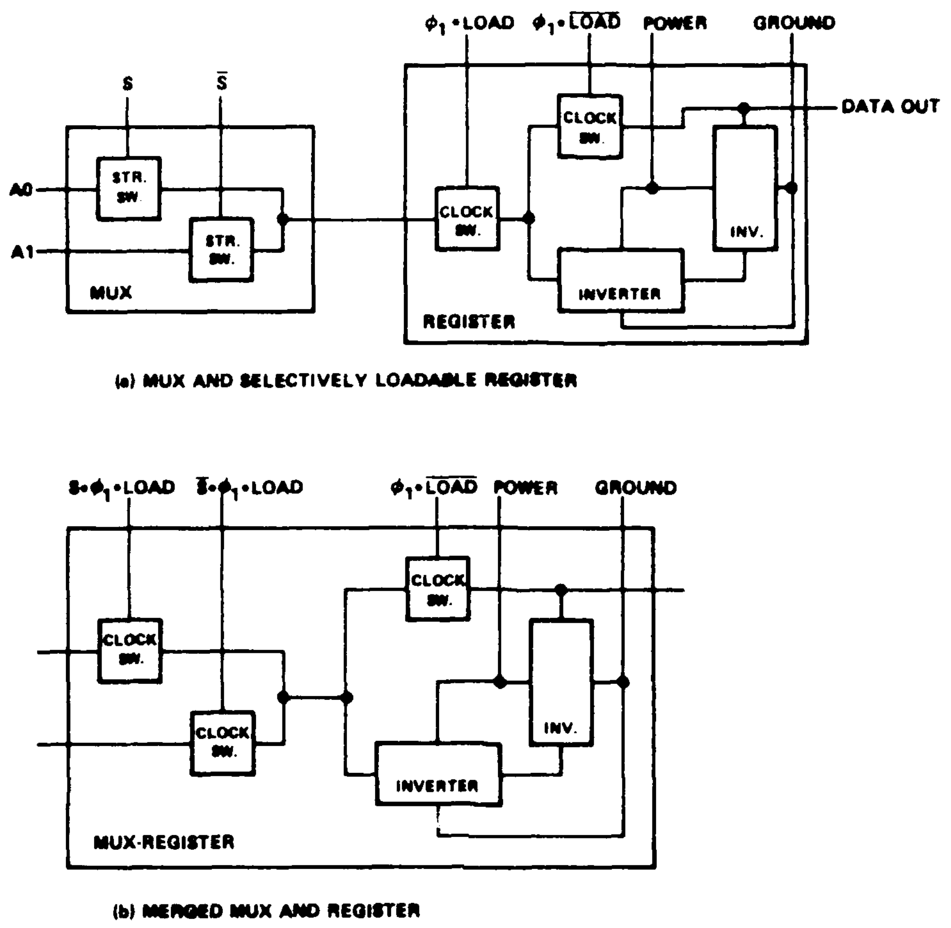

$\phi_1 \cdot$LOAD
$\phi_1 \cdot \overline{\text{LOAD}}$
POWER
GROUND
S
$\overline{S}$
DATA OUT
STR. SW.
A0
A1
CLOCK SW.
INV.
INVERTER
MUX
REGISTER
(a) MUX AND SELECTIVELY LOADABLE REGISTER
$S \cdot \phi_1 \cdot$LOAD
$\overline{S} \cdot \phi_1 \cdot$LOAD
MUX-REGISTER
(b) MERGED MUX AND REGISTER

Figure 6. An example of structure merging.

The forms of structure sharing illustrated above are the only ones admitted by our design paradigm. This constraint 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 the component. Thus, components sharing a component can be independently edited and [illegible] The relationship between a merged component and the components which it merges is also maintained by Palladio. In particular, maintaining such a relationship permits verification (by simulation) that the merged component or its refinements achieve the combined functionality of the components which it merges.

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 physical realizability. Circuit verification is usually performed by simulation*; which, in Palladio, means modeling the circuit's structure in the computer, specifying an initial state for the circuit, and 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 is expressed as perspective-specific *rules* that are triggered by changes in a component's state and that, in turn, change the state of the circuit. A unidirectional pass transistor has three ports: IN, OUT, and CTL. An example of a 3-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 $\times$ 3

*There is some current work on using formal methods for behavioral verification of circuits; see, for example, Barrow[16] level-strength logic.[12] The following three 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 via Rules

In an integrated design environment, specification of behavior must take a fairly flexible form, as it must serve many diverse purposes:

Purpose 1. *The behavioral specification is part of the overall specification of the design and must be in a form the designer can both enter and (at some later point) comprehend.*

The rule format has the advantage of being fairly transparent and well structured.

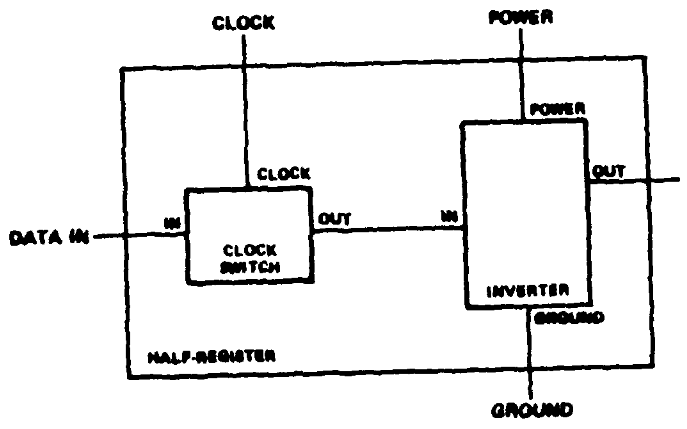
Purpose 2. *The behavioral specification must be usable by a simulator; furthermore, connected components whose behavioral specifications are at different perspectives should be simulatable.*

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 for 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)*

Purpose 3. *A component's behavioral specification must be compatible with its structural specification.*

A component's behavioral specification can be compared, by simulation, with the behavioral specifications of its interconnected components, to verify the correctness of the component decomposition. For example, in Figure 7, the HALF-REG component could be verified by simulations using its indicated behavioral specification and using the behavior it derives from its components.



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.

Purpose 4. *The behavioral specification may serve as a constraint or as input for other programs, for example, an automatic refinement program.*

For instance, behavior described from a Boolean logic perspective could be used as input to a PLA generator to produce a layout perspective design. Rules express behavior in a form that is convenient to use as input to other programs.

Purpose 5. *It should be possible to specify behavior for "dummy components" that are not implemented, but are only used for generating or monitoring signals during a simulation.*

Such dummy components are analogous to "PRINT" statements that are used to help debug a software program but are removed afterwards

Purpose 6. *The behavioral specification should be usable for providing explanations for a particular simulation result.*

It is easy to produce primitive explanations from records of rule activations.[3]

Alternatively, programming language procedures could have been 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 upon MARS[17] (Multiple Abstraction Rule-based Simulator), a general-purpose, event-driven simulator whose generality derives, in part, from the logic reasoning system, MRS[5], in which it is implemented. A logic reasoning system contains, as data, a set of assertions and a collection of rules triggered by the presence of assertions and capable of producing new assertions; a logic reasoning system uses *inference rules* to control the manner in which new assertions are added to the current set of assertions. MARS expresses the state of a circuit as a set of assertions and maps the behavioral rule formalism described earlier into MRS's rule formalism in a straightforward way. Repeated use of the inference rule *modus ponens* (i.e., 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 the highly detailed perspectives only when it is necessary to verify the design from that perspective, the component hierarchy can be exploited to achieve large gains in simulation performance. In a typical hierarchical simulation, the majority of components are being simulated from their high-level behavioral perspective, either out of necessity (as their structure has not yet 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. For example, for a logic simulation various state attribute values can be denoted by distinct colors. This dynamic display produces a "movie" of the circuit's behavior. The state history of a simulation can also be saved in a formatted text file for later analysis.

Implementation

Palladio is implemented using object-oriented, data-oriented, rule-based, and logic-language programming paradigms.

The object-oriented, data-oriented, and rule-based facilities are largely provided by the *LOOPS* (Lisp Object-Oriented Programming System) programming environment.[4] LOOPS was developed by Daniel Bobrow and Mark Stefik and 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 logic language system, *MRS* (Metalevel Reasoning System),[5] is based on the predicate calculus and includes full logic inference capabilities. MRS was developed by Michael Genesereth. LOOPS' rule facilities and MRS each provide Palladio with the mechanisms for developing, integrating, and testing knowledge-based expert system design aids.

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

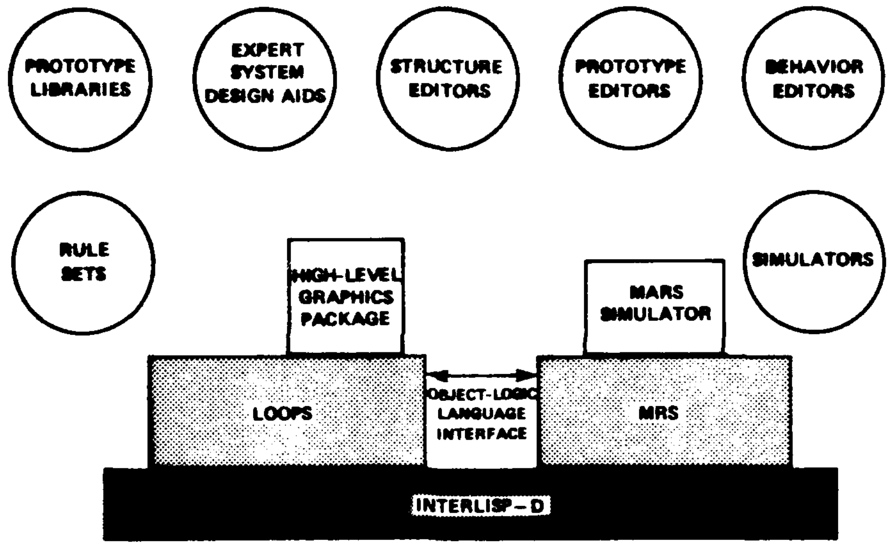


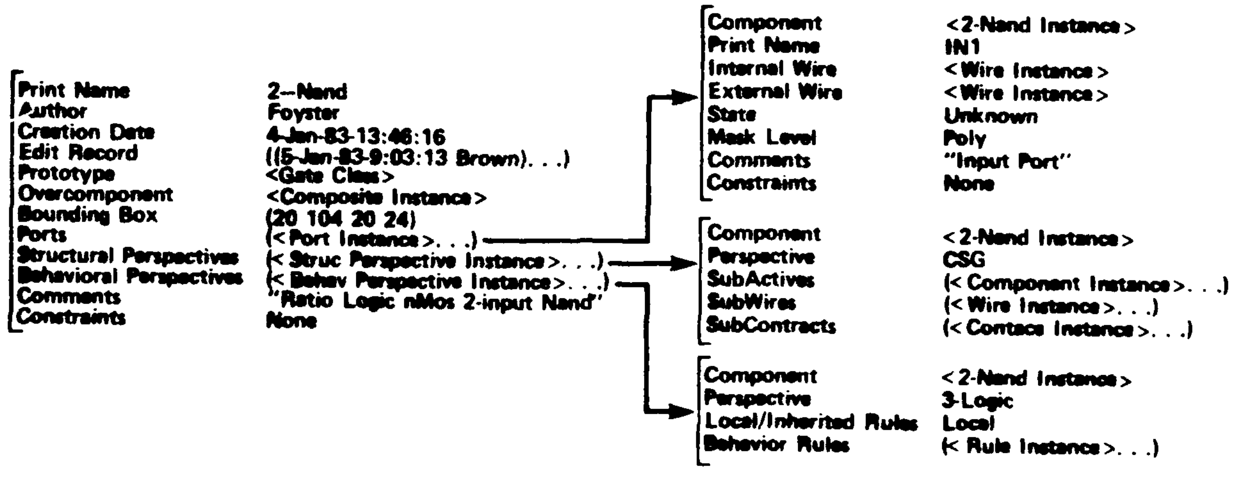
Figure 8. Palladio system architecture.

Object-oriented Programming

Palladio is mainly implemented in an *object-oriented programming* paradigm whose basic entities are objects, classes, messages, and methods (see, e.g., *Smalltalk*[19]). Every major system component in Palladio (e.g., structure editors, behavior editors, and simulators) is represented as an *object*, that is, a package that includes a data structure and a set of *methods* (i.e., procedures) for operating on that structure. Every circuit entity in Palladio is also an object (e.g., circuit, components, ports, and wires).

Large classes of objects often share identical data structures, differing only in the values of those data structures (e.g., all wires have two ends and an associated signal type, signal strength, etc.). Object-oriented languages exploit this property with *classes*, special objects that define a basic data structure and its associated methods; a class serves as a template for creating its *instances*--those objects sharing the same data structure as their class. In Palladio, a prototype component is defined by a class object, and any instance of the prototype occurring in a circuit specification is an instance object of the prototype class object.

The data structure part of a LOOPS object is a frame composed of attribute-value pairs. The frame of a 2-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 (e.g., atom, list, array, object, active value); the values of some of the attributes in the example 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.



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	(< Contacs Instance>. . .)

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

Figure 9. 2-input NAND gate frame.

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.

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 Pers[illegible]ive 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.

An instance of the class Wire is shown in Figure 11.

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.

Object-oriented programming provides a powerful, flexible solution to the problem of representing generic actions; every type of entity provides its own definition for a generic action such as displaying itself on the screen. A *message* sent to an object results in the invocation of the method 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, respectively; 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, an ADD message is sent to the prototype. The message-passing technique is a natural means for creating software modularity. Also, the sender of a message only need know that a particular recipient can respond to a particular message and not how that recipient will respond; the recipient of the message knows the appropriate method and how to invoke it.

Palladio's Class Inheritance Network.

Object-oriented programming uses the class/instance distinction to exploit the fact that large classes of objects share identical data structures. Object-oriented programming also exploits the fact that classes often have *similar*, but not identical, data structures. 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; furthermore, the default values of its inherited attributes and the associated methods of its inherited messages can differ from those of its parents.

Part of the class inheritance network for Palladio's circuit objects is shown in Figure 12.

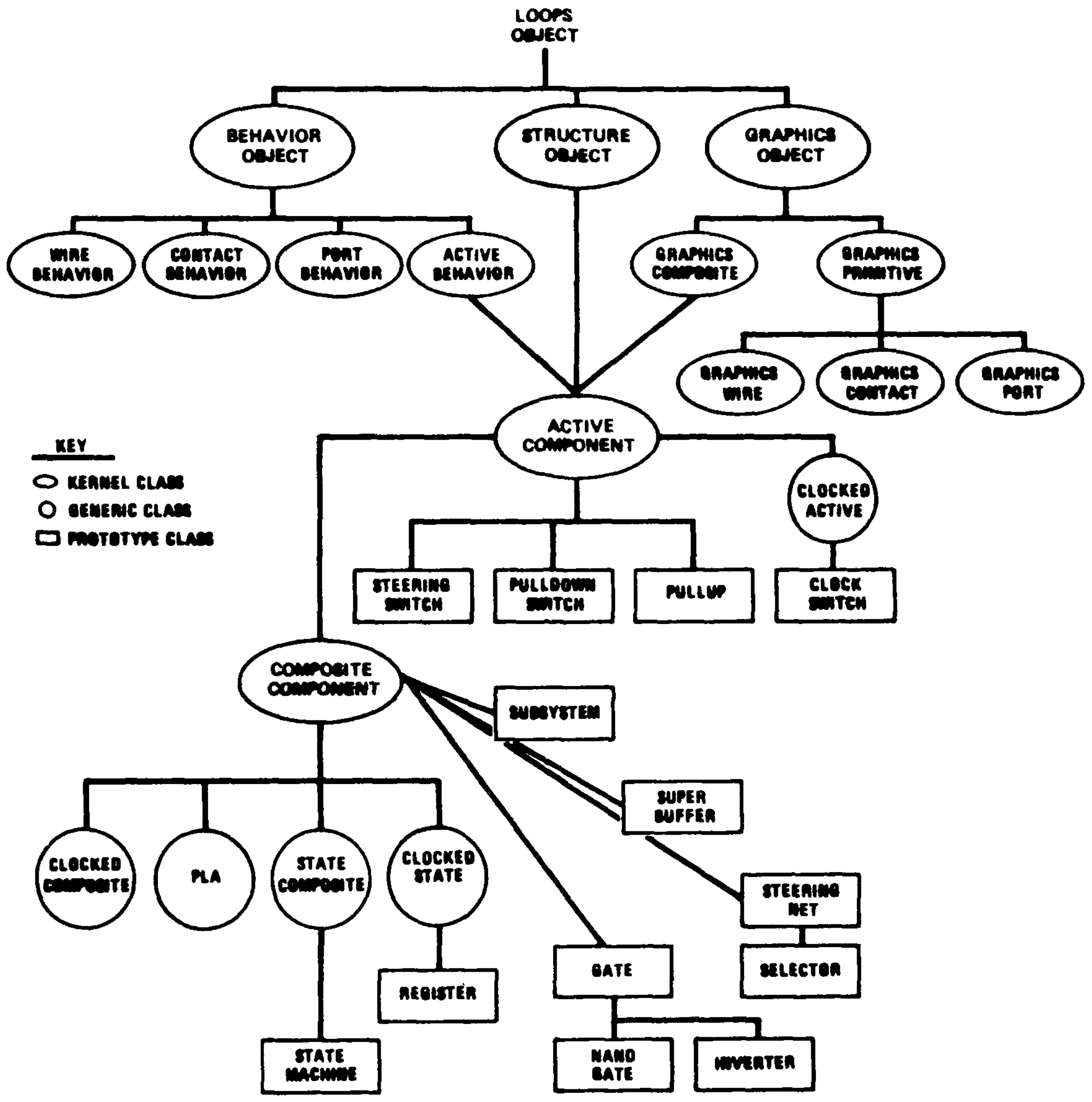


Figure 12. Partial class inheritance network.

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 (e.g., 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 3-input NAND gate prototype class could be created as a subclass of either the Gate class or the NAND class.

Palladio provides interactive graphics editors for defining new prototype components. The definition of a new perspective, however, can invol.e the creation of new generic classes that have attributes and methods not inherited from their kernel parents. Currently, such classes are created with the LOOPS object editor, which requires familiarity with the underlying object representations and Interlisp.

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, φ1, 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 are still unspecified is indeterminate. Eventually, the design becomes sufficently detailed so 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 value of the signal type attribute on all wires; whenever that value is set, the active value's procedure propagates the signal type throughout the wire's 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. These design aids 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, the Palladio environment 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 formulating 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 refinement steps in deriving a layout is to assign mask levels to the interconnect (wires and contacts) between the components. Given a design tool that could perform this task, a designer can quickly see the consequences of circuit refinement and avoid introducing unnecessary errors.

The goal of wire assignment 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 *C1* is attached to a wire with a particular mask level, the mask level assigner tries to maintain the already established mask level assignment when considering a component *C2*, which contains *C1*.

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 introducing any unintended connections between intersecting wires or any inadvertant transistors (by intersecting polysilicon and diffusion).

To keep the problem manageable, a constraint was introduced: *The planar topology of the interconnect must remain as given in the CSG perspective*; that is, 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 using Lisp procedures for the base-level actions, and LOOPS rules to control usage of the Lisp procedures. There are two base level actions: *(1) assignment of a mask level to a wire* and *(2) introduction of vias for changing 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;

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 strategy 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 one 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 no unintentional shorts or transistors have been produced.

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.

```
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 wire mask assigner 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, ranking of intersections uses a set of rules to determine the relative importance of each intersection. This rule set requires such global information 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 contributing to the high performance and the 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 rule set can test the attribute values of the object and conditionally execute procedures, set attributes values, or send messages. When a rule set is invoked, the rules are tested and the conditional actions are executed.

The ***WireAssigner*** is an object created to carry out the mask level assignment task. A ***WireAssigner*** responds to the following messages:

1. The *Activate* message takes a circuit as a parameter and initializes the assignment process.

2. The *FindIntersections* message finds all intersections between wires in the circuit under consideration. The method for this message creates a new ***Intersection*** object for each intersection found and stores the objects in a list.

3. The *ReorderIntersections* message sorts the intersection object list according to the importance of the intersections as determined by a set of rules.

4. The *MakeAssignment* message invokes the appropriate base level actions which perform the actual assignment of a mask level (including introducing vias) to a wire.

Intersection objects respond to the messages *MaskConflict?* and *ResolveConflict.* The

ResolveConflict message 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. These rule sets determine the priority of two conflicting wires and the costs 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 the costs of various mask levels. When the cost values have been established, the *GenerateMinAssignment* message 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 the *MakeAssignment* message to the ***WireAssigner***, and the assignment is made. The wire assignment process can be dynamically displayed on the color screen.

Infinite loops in the wire assignment process could occur because a prior assignment to a wire 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 consisting of 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 only a small number of rules: the resulting 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 in a more intelligent manner 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: The performance of an expert system can be incrementally improved by the addition of more knowledge. Also, 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 which 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 envoked from within circuit design editors. This allows a designer to easily augment a prototype component library during the circuit design process.

2. A behavioral rule editor wl ch gives syntactic support for entering and modifying behavioral specifications of both prototype components and circuit components.

3. An event-driven simulator which uses the behavioral and structural specifications of a circuit to simulate it. The simulator can perform hierarchical simulation (i.e., use either the specified behavior of a component or the behavior induced by the behavior of 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 3-value logic perspective and some of which have behavior specified from a 3 $\times$ 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

which we have found to be 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. These mechanisms are largely provided by LOOPS and MRS.

Our initial (and current) implementation of Palladio was a research effort. Our interest was in in investigating a set of concepts about circuit design environments. We have used Palladio to design several circuits using perspectives ranging from architectural through cell-based sticks diagram levels. We feel that even this limited expeience has substantiated the utility of many of Palladio's underlying concepts, for example, the hierarchical use of multiple perspectives, distinct structural and behavioral perspectives, behavioral specifications using a rule format, a behavioral simulator applicable to all levels of behavioral specification, and design aids implemented as rule-based expert systems. However, in its current implementation, Palladio is difficult to use effectively by anyone other than its builders, it is not particularly robust, and it has significant efficency problems (e.g., about eight hours on a Xerox 1100 computer to run a 1000 event simulation of the multi-processor circuit described on page 10).

Palladio's Performance

During the implementation of Palladio we were often uncertain as to exactly what system capabilities would prove to be useful. Thus, whenever we were faced with a flexibility versus efficency trade-off, we opted for flexibility. We have paid a price for this 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. We have found that some of the capabilities of the system are of very limited utility, for example, the completely general underlying representational mechanisms for circuit structure and behavior. We are currently reimplementing Palladio observing more realistic flexibility versus efficiency trade-offs. This reimplementation should result in an overall order-of-magnitude performance improvement. However, even with such improvments and running on the more powerful Xerox 1132 computer we estimate that we would be limited to circuits with at most tens-of-thousands of low-level components. Our conjecture is that flexible, fully-integrated design environments for custom, VLSI-scale circuits will require computers more powerful than those that are currently available.

Conclusions

The Palladio system is an exploratory design environment that 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 Palladio, 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; this requires *representation* of the tools and languages in an easily modifiable and augmentable form.

Table 2 is a summary of the ways in which we have used different programming paradigms for building different elements of the design environment.

Table 2. Programming Paradigm Applications.

Programming paradigm	***Design environment application***
Rule-based	Incrementally constructed expert design aids
Object oriented	Design specification
Logical language	Simulation
Data oriented	Constraint propagation

Multiple programming paradigms have proved useful for explicitly representing diverse kinds of tools and languages and for making their modification and extension as straightforward and rapid as possible. The table is meant to suggest only a few preliminary correspondences; finding best fits between programming paradigms and design environment applications is a novel and worthwhile area for research. Palladio is *an exploratory design environment that contains an exploratory programming environment* in order to experiment easily with varying elements of an integrated design environment.

Further research areas we are actively pursuing include the design of a language that spans the spectrum of functionality, behavior, and structure, thus eliminating some of the parallel languages required, and the design of a language in which circuit design problems (and theories of circuit design) can be stated, based on the assumption that the circuit design problem and the circuit design co-evolve. Basic terms of such a language include design goals, tasks, constraints, and trade-offs.

Palladio is an early attempt to explore the stuff on which circuit design environments are built.

Acknowledgements

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 Allan Bell, Daniel G. Bobrow, Lynn Conway, and Mark Stefik at Xerox Palo Alto Research Center (PARC) 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, March/April 1983, pp. 46-55.

2. D. S. Nau, "*Expert Computer Systems,*" *Computer,* Vol. 16, No. 2, Feb. 1983, pp. 63-85.

3. F. Hayes-Roth, D. Waterman and D. Lenat (Eds.) "*Building Expert Systems,*" Addison-Wesly, Inc., Reading Mass., 1983.

4. D. G. Bobrow and M. Stefik, "*The LOOPS Manual,*" Memo KB-VLSI-81-13, Xerox Palo Alto Research Center, Aug. 1981, revised Aug. 1982.

5. M. Genesereth, "*MRS: A Metalevel Representation System,*" Stanford University, Heuristic Programming Project Working Paper HPP-83-28, June 1983.

6. A. Barr and E. Feigenbaum, "*The Handbook of Artificial Intelligence,*" Vol. 1, Ch. 3, William Kaufman, Inc., Los Altos, Calif., 1982.

7. C. U. Smith and J.. A. Dallen, "*A Comparison of Design Strategies for Software and VLSI,*" *Proc. Spring Compcon 83,* Feb. 28-March 3, 1983, pp. 263-268.

8. W. Teitelman, "*INTERLISP Reference Manual.*" Xerox Corporation, Palo Alto Research Center, 1978.

9. B. A. Shiel, "*Power Tools for Programmers,*" *Datamation,* Feb. 1983, pp. 131-144.

10. H. Simon, "*The Sciences of the Artificial,*" 2nd edition, MIT Press, Cambridge, Mass., p. 148.

11. M. Stefik, D. Bobrow, A. Bell, H. Brown, L. Conway, and C. Tong, "*The Partitioning of Concerns in Digital Systems Design,*" *Proc. Conference on Advanced Research in VLSI,* P. Penfield (Ed.), Artech House, Jan. 1982.

12. R. E. Bryant, "*A Switch-level Model and Simulator for MOS Digital Systems,*" Caltech Technical Report 5065, 1983.

13. C. Mead and L. Conway, "*Introduction to VLSI Systems,*" Addison-Wesley Publishing Company, 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.

16. H. G. Barrow, "*Proving the Correctness of Digital Hardware Designs,*" *Proc. of AAAI-83,* William Kaufman, Inc., Los Altos, Calif, 1983.

17. N. Singh, "*MARS: A Multiple Abstractions Rule-Based Simulator,*" Fairchild Laboratory for Artificial Intelligence Research Technical Report 17, 1983.

18. B. A. Sheil and L. M. Masinter (Eds.) "*Cognitive and Instructional Sciences Series CIS-5,*" Xerox Palo Alto Research Center, Sept. 1980, revised Jan. 1983.

19. A. Goldberg, "*Introducing the Smalltalk-80 System,*" *Byte,* 6:8, Aug. 1981.

20. B. G. Buchanan and R. O. Duda, "*Principles of Rule-Based Expert Systems,*" in *M. Yovits (Ed.) Advances in Computers,* Vol. 22, Academic Press, New York, 1983.